HZ BOOKS
华章科技

C++标准委员会成员和IBM XL编译器中国开发团队共同撰写，权威性毋庸置疑

系统、深入、详尽地讲解了C++11新标准中的新语言特性、新标准库特性、对原有特性的改进，以及所有这些新特性的应用

华章原创精品



深入理解C++11

C++11新特性解析与应用

Understanding C++11

Analysis and Application of New Features

（加）Michael Wong
IBM XL编译器中国开发团队 著



机械工业出版社
China Machine Press



Understanding C++11

Analysis and Application of New Features

从C++98到C++11，C++标准10年磨一剑，相比C++98/03，C++11则带来了数量可观的变化，这包括了约140个新特性，以及对C++03标准中约600个缺陷的修正，这使得C++11更像是从C++98/03中孕育出的一种新语言。相比较而言，C++11能更好地用于系统开发和库开发、更易于教学（语法更加泛化和简单化）、更加稳定和安全，不仅功能变得更强大，而且能提升程序员的开发效率。随着各种编译器对C++11标准的支持更加完善，C++11标准势必会逐渐成为主流，作为一个有经验的C++程序员，如果你想比别人先行一步，抢占先机，那么本书为你提供了绝好的机会，它系统、深入、详尽地讲解了C++11新标准中的新语言特性、新标准库特性、对原有特性的改进，以及所有这些新特性的应用。

本书主要内容：

- C++11标准的设计目标和设计原则，以及C++11新特性预览和分类；
- 出于保证稳定性和兼容性而增加的新特性，如long long整数类型、静态断言、外部模板，等等；
- 具有广泛可用性、能与其他已有的或者新增的特性结合起来使用的、具有普适性的一些新特性，如继承构造函数、委派构造函数、列表初始化，等等；
- 对原有一些语言特性的改进，如auto类型推导、追踪返回类型、基于范围的for循环，等等；
- 在安全方面所做的改进，如枚举类型安全和指针安全等方面的内容；
- 为了进一步提升和挖掘C++程序性能和让C++能更好地适应各种新硬件的发展而设计的新特性，如多核、多线程、并行编程，等等；
- 颠覆C++一贯设计思想的新特性，如lambda表达式等；
- C++11为了解决C++编程中各种典型实际问题而做出的有效改进，如对Unicode的深入支持等；
- C++11标准与其他相关标准的兼容性和区别、C++11中弃用的特性、编译器对C++11的支持情况，以及学习C++11的相关资源。

11

上架指导：计算机/程序设计/C++

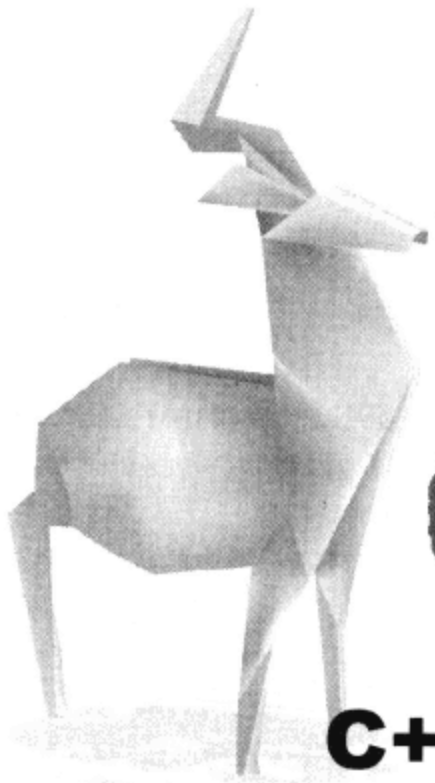
ISBN 978-7-111-42660-8

9 787111 426608 >

定价：69.00元

客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259
投稿热线：(010) 88379604

数字阅读：www.hzmedia.com.cn
华章网站：www.hzbook.com
网上购书：www.china-pub.com



深入理解C++11

C++11新特性解析与应用

Understanding C++11

Analysis and Application of New Features

（加）Michael Wong
IBM XL编译器中国开发团队 著



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

深入理解 C++11 ：C++11 新特性解析与应用 / Michael Wong, IBM XL 编译器中国开发团队著 . —北京 ：机械工业出版社，2013.6
（原创精品系列）

ISBN 978-7-111-42660-8

Ⅰ. 深… Ⅱ. ① M… ② I… Ⅲ. C 语言 – 程序设计 Ⅳ. TP312

中国版本图书馆 CIP 数据核字（2013）第 110482 号

本书法律顾问　北京市展达律师事务所
封底无防伪标均为盗版
版权所有 • 侵权必究
©Copyright IBM Corp.2013

国内首本全面深入解读 C++11 新标准的专著，由 C++ 标准委员会代表和 IBM XL 编译器中国开发团队共同撰写。不仅详细阐述了 C++11 标准的设计原则，而且系统地讲解了 C++11 新标准中的所有新语言特性、新标准库特性、对原有特性的改进，以及如何应用所有这些新特性。

全书一共 8 章 ：第 1 章从设计思维和应用范畴两个维度对 C++11 新标准中的所有特性进行了分类，呈现了 C++11 新特性的原貌 ；第 2 章讲解了在保证与 C 语言和旧版 C++ 标准充分兼容的原则下增加的一些新特性 ；第 3 章讲解了具有广泛可用性、能与其他已有的或者新增的特性结合起来使用的、具有普适性的一些新特性；第 4 章讲解了 C++11 新标准对原有一些语言特性的改进，这些特性不仅能让 C++ 变得更强大，还能提升程序员编写代码的效率 ；第 5 章讲解了 C++11 在安全方面所做的改进，主要涵盖枚举类型安全和指针安全两个方面的内容；第 6 章讲解了为了进一步提升和挖掘 C++ 程序性能和让 C++ 能更好地适应各种新硬件的发展而设计的新特性，如多核、多线程、并行编程方面的新特性 ；第 7 章讲解了一些颠覆 C++ 一贯设计思想的新特性，如 lambda 表达式等 ；第 8 章讲解了 C++11 为了解决 C++ 编程中各种典型实际问题而做出的有效改进，如对 Unicode 的深入支持等。附录中则介绍了 C++11 标准与其他相关标准的兼容性和区别、C++11 中弃用的特性、编译器对 C++11 的支持情况，以及学习 C++11 的相关资源。

机械工业出版社（北京市西城区百万庄大街 22 号　　邮政编码　100037）
责任编辑：孙海亮
三河市杨庄长鸣印刷装订厂印刷
2013 年 6 月第 1 版第 1 次印刷
186mm × 240 mm • 20.5 印张
标准书号：ISBN 978-7-111-42660-8
定　　价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

免责声明

本书言论仅为作者根据自身实践经验所得，仅代表个人观点，不代表 IBM 公司的官方立场和看法，特此声明。

IBM XL 编译器中国开发团队简介

IBM 拥有悠久的编译器开发历史（始于 20 世纪 80 年代），在全球拥有将近 400 名高素质工程师组成的研发团队，其中包括许多世界知名的研究学者和技术专家。IBM 一直以来都是编程语言的制定者和倡导者之一，并将长期在编译领域进行研发和投资。IBM XL 编译器中国开发团队于 2010 年在上海成立，现拥有编译器前端开发人员（C/C++）、后端开发人员，测试人员，以及性能分析人员等。团队与 IBM 北美编译器团队紧密合作，共同开发、测试和发布基于 POWER 系统的 AIX 及 Linux 平台下的 XL C/C++ 和 XL Fortran 系列产品，并对其提供技术支持。虽然团队成立时间不长，但已于 2012 年成功发布最新版本的 XL C/C++ for Linux V12.1 & XL Fortran for Linux V14.1，并获得 7 项发明专利。团队成员拥有丰富的编译器开发经验，对编译技术、编程语言、性能优化和并行计算等都有一定的研究，也对 C++11 标准的各种新特性有较早的研究和理解，并正在实际参与 C++11 新特性的开发工作。

欢迎广大读者关注 IBM XL 编译器中国开发团队博客：http://ibm.co/HK0GCx，及新浪微博：www.weibo.com/ibmcompiler，并与我们一起学习、探讨 C++11 和编译技术。

作者个人简介

官孝峰　毕业于上海交通大学软件学院。多年致力于底层软件开发，先后供职于 AMD 上海研发中心，IBM 上海研发中心。在嵌入式系统及应用、二进制翻译、图形驱动等领域有丰富的实践经验。2010 年加入 IBM XL 中国编译器中国开发团队，负责 XL C/C++/Fortran 编译器后端开发工作，专注于编译器后端优化、代码生成，以及语言标准等领域。在 C++11 标准制定后，率先对标准进行了全面深入的研究，并组织团队成员对新语言标准进行学习探讨。是数项国内外专利的发明人，并曾于 DeveloperWorks 社区发表英文论文一篇。

陈晶　毕业于西安交通大学通信与信息工程专业，有多年的编译器文档开发写作经验。2008 年起供职于 IBM 上海研发中心，一直致力于研究 C++11 标准的各项新特性，并负责该部分的技术文档撰写。精通 C/C++ 语言，对编译器领域有浓厚的兴趣。负责工作部门内部的编译器产品技术培训。在 DeveloperWorks 社区发表过多篇论文。拥有一项国内专利。

任剑钢　毕业于复旦大学计算机专业。2010 年加入 IBM XL 中国编译器中国开发团队，先后从事 XL Fortran 编译器前端、XL C/C++/Fortran 编译器后端等各种开发工作。对于技术敏感而热衷，擅长 C/C++/Java 等多种编程语言，现专注于编译器代码优化技术。拥有一项专利，并带领团队在 IBM Connections 平台的技术拓展大赛中赢得大奖。

朱鸿伟　毕业于浙江大学计算机科学与技术专业。资深软件开发工程师，有多年系统底层软件开发和 Linux 环境开发经验。一直致力于 C/C++ 语言、编译器、Linux 内核等相关技术的研究与实践，关注新技术和开源社区，对于 Linux 内核以及 Linux 平台软件的开发有着浓厚的兴趣，曾参与 Linux 开源软件的开发与设计。2010 年加入 IBM XL 中国编译器中国开发团队，现专注于编译器前端技术研究与开发工作。

张青山　毕业于福州大学计算机系。从事嵌入式开发多年，曾致力于 Linux 内核和芯片驱动程序的开发、及上层应用程序的编写。2010 年加入 IBM XL 编译器中国开发团队，负责 XL C++ 编译器前端的研发工作。对 C99、C++98、C++11 等语言标准及编译理论有深入理解，

并实际参与 C++11 前端各种特性的实现。此外还致力于编译器兼容性的研究和开发。

纪金松 中国科学技术大学计算机体系结构专业博士，有多年编译器开发经验。2008 年起先后就职于 Marvell（上海）、腾讯上海研究院、IBM 上海研发中心。一直致力于系统底层软件的研究与实践，在编译器后端优化、二进制工具链、指令集优化、可重构计算等相关领域有丰富的实战经验。在国内外杂志和会议中发表过 10 多篇论文，并拥有多项国内外专利。

郭久福 毕业于华东理工大学控制理论与控制工程专业。拥有近 10 年的系统软件开发经验，曾就职于柯达开发中心、HP 中国以及 IBM 中国软件实验室。对 C 语言标准以及 C++ 语言标准有深入研究，近年来致力于编译器前端的开发和研究。

林科文 毕业于复旦大学计算机软件与理论专业，有多年底层系统开发经验。2010 年起供职于 IBM 上海研发中心，现从事编译器后端开发工作。一直致力于系统软件的研究和实践，以及编译器代码优化等领域。活跃于 DeveloperWorks 社区，是三项国内专利的发明人。

班怀芸 毕业于南京理工大学计算机应用技术专业。资深测试工程师，在测试领域耕耘多年，曾任职于 Alcatel-Lucent 公司，有丰富的项目经验。2011 年加入 IBM XL 编译器中国开发团队，现从事和 C/C++ 语言相关的测试领域研究，负责用编译器实现相关需求分析、自动化测试方案制定及实现等工作。持续关注语言和测试技术的发展，对 C/C++ 新特性和语言标准有较深的理解。

蒋健 毕业于复旦大学计算机科学系，资深编译器技术专家。先后供职于 Intel、Marvell、Microsoft 及 IBM 等各公司编译器开发部门，参与并领导业界知名的编译器后端多种相关研发工作，并且拥有近十项编译器方面的专利。现负责 XL C/C++/Fortran 编译器后端开发，并领导 IBM XL 中国编译器中国开发团队各种技术工作。

宋瑞 毕业于北京大学微电子学院，在软件测试领域工作多年，对于测试架构以及软件的发布流程具有丰富的经验，曾就职于 Synopsys 和 Apache design solution。2011 年加入 IBM XL 编译器中国开发团队，从事和编译器、C/C++ 语言相关的研究与测试，对 C/C++ 新语言特性和标准有较深的理解和研究。

刘志鹏 毕业于南京理工大学计算机应用专业，2010 年加入 IBM XL 中国编译器中国开发团队。先后从事 Fortran、C++ 前端的开发工作。现致力于 C++ 语言新标准、语言兼容性等研究与开发。对编译器优化技术、Linux 内核开发有浓厚兴趣，擅长 C/C++/Java 等多种语言。

毛一赠 毕业于上海交通大学，现任职于 IBM 编译器中国开发团队，从事 XL C/C++ 编译器前端的开发工作，在此领域有多年研究经验。具有丰富的实际项目经验并持续关注语言发展。擅长 C++、C、Java 等语言，对 C++ 模板有深入研究。此外对 C#/VB/perl/Jif/Fabric 等语言也均有所涉猎。热爱编程与新技术，活跃于 DeveloperWorks 等社区，发表过技术博客数篇。

张寅林 毕业于上海交通大学信息安全工程专业。2010 年起加入 IBM XL 编译器中国开发团队，专注于编译器后端性能优化开发工作。对于编译器优化算法，Linux 操作系统体系结构与实现有浓厚的兴趣，擅长 C/C++/Python 编程语言。目前从事 IBM 企业级存储服务器的开发工作。

刘林 东南大学计算机科学与工程学院硕士，有多年底层系统开发经验。2010—2012 年间就职于 IBM XL 编译器中国开发团队，先后从事编译器测试及后端开发工作。对嵌入式、Linux 操作系统体系结构与实现有浓厚的兴趣。

Preface

If you are holding this book in the store, you might be wondering why you should read this book among many C++ books. First, you should know this book is about the latest new C++11 (codenamed C++0x) Standard ratified at the end of 2011. This new Standard is almost like a new language, with many new language and library features but there is a strong emphasis for compatibility with the last C++98/03 Standard during design. At the time of printing of this book in 2013, this is one of the first few C++11 books published. All books that do not mention C++11 will invariably be talking about C++98/03.

What makes this book different is that it is written by Chinese writers, in its original Chinese language. In fact, all of us are on the C++ compiler team for the IBM xlC++ compiler, which has been adding C++11 features since 2008.

For my part, I am the C++ Standard representative for IBM and Canada and have been working in compilers for 20 years, and is the author of several C++11 features while leading the IBM C++ Compiler team.

For C++ users who read Chinese, many prefer to read an original Chinese language book, rather than a translated book, even if they can read other languages. While very well written also by experts from the C++ Standard Committee, these non-Chinese books' translation can take time, or result in words or meanings that are loss in translation. The translator has a tough job as technical books contain many jargon and new words that may not have an exact meaning in Chinese. Different translators may not use the same word, even within the same book. These are reasons that lead to a slow dissemination of C++ knowledge and slows the adoption of C++11 in Chinese.

These are all reasons that lead to weak competitiveness. We aim to improve that competiveness with a book written by Chinese-speaking writers, with a uniform language for jargons, who understand the technology gap as many of the writers work in the IBM Lab in Shanghai. We know there are many Chinese-speaking C++ enthusiasts who are eager to learn and use the updates to their favorite language. The newness of C++11 also demands a strong candidate in the beginner to intermediate level of C++11, which is the level of this book.

You should do well reading this book, if you are:

- an experienced C programmer who wants to see what C++11 can do for you.
- already a C++98/03 programmer who wants to learn the new C++11 language.
- anyone interested in learning the new C++11 language.

We structure this book using the design principles that Professor Bjarne Stroustrup, the father of C++ followed in designing C++11 through the Standard Committee. In fact, we separated this book into chapters based on those design principles. These design principles are outlined in the first chapter. The remaining chapters separate every C++11 language features under those design classifications. For each feature, it will explain the motivation for the feature, the rules, and how it is used, taking from the approved C++11 papers that proposed these features. A further set of appendices will outline the current state of the art of compiler support for C++11, incompatibilities, deprecated features, and links to the approved papers.

After reading this book, you should be able to answer questions such as:

- ❑What is a lambda and the best way to use it?
- ❑How is decltype and auto type inference related?
- ❑What is move semantics and how does it solve the forwarding problem?
- ❑I want to understand default and deleted as well as explicit overrides.
- ❑What did they replace exception specifications with and how does noexcept work?
- ❑What are atomics and the new memory model?
- ❑How do you do parallel programming in C++11?

What we do not cover are the C++11 changes to the Standard library. This part could be a book itself and we may continue with this as Book II. This means we will not talk about the new algorithms, or new class libraries, but we will talk about atomics since most compilers implement atomics as a language feature rather than a library feature for efficiency reason. However, in the C++11 Standard, atomics is listed as a library feature simply because it could be implemented at worst as a library, but very few compilers would do that. This book is also not trying to teach you C++. For that, we particularly recommend Professor Stroustrup's book Programming principles & Practice Using C++ which is based on an excellent course he taught at Texas A&M University on programming.

This book could be read chapter by chapter if you are interested in every feature of C++11. More likely, you would want to learn about certain C++11 feature and want to target that feature. But while reading about that feature, you might explore other features that fall under the same design guideline.

We hope you find this book useful in your professional or personal learning. We learnt too during our journey of collaborating in writing this book, as we wrote while building the IBM C++ compiler and making it C++11 compliant.

The work of writing a book is long but it is well worth it. While I have been thinking about writing this book while working on the C++ Standard, it was really Xiao Feng Guan who motivated me to start really stop thinking and start doing it for real. He continued to motivate and lead others through their writing assignment process and completed the majority of the work of organizing

this book. I also wish to thank many who have been my mentors officially and unofficially. There are too many to mention but people such as Bjarne Stroustrup, Herb Sutter, Hans Boehm, Anthony Williams, Scott Meyers and many others have been my teachers and great examples of leaders since I started reading their books and watching how they work within large groups. IBM has generously provided the platform, the time, and the facility to allow all of us to exceed ourselves, if only just a little to help the next generation of programmers. Thank you above all to my family Sophie, Cameron, Spot the Cat, and Susan for lending my off-time to work on this book.

Michael

序

当你在书店里拿起这本书的时候，可能最想问的就是：这么多 C++ 的书籍，为什么需要选择这一本？回答这个问题首先需要知道的是，这是一本关于在 2011 年年底才制定通过的 C++11（代码 C++0x）的新标准的书籍。这个新标准看起来就像是一门新的语言，不仅有很多的新语言特性、标准库特性，而且在设计时就考虑了高度兼容于旧有的 C++98/03 标准。在 2013 年出版的 C++ 的书籍中，本书是少数几部关于 C++11 的书籍之一，而其他的，则会是仅讲解 C++98/03 而未提及 C++11 的书籍。

相比于其他书籍，本书还有个显著特点——绝大多数章节都是由中国作者编写。事实上，本书所有作者均来自 IBM XL C++ 编译器开发团队。而团队对于 C++11 新特性的开发，早在 2008 年就开始了。

而我则是一位 IBM 和加拿大的 C++ 标准委员会的代表。我在编译器领域已工作了 20 多年。除了是 IBM C++ 编译器开发团队的领导者之外，还是一些 C++11 特性的作者。

对于使用中文的 C++ 用户而言，很多人还是喜欢阅读原生的中文图书，而非翻译版本，即使是在他们具备阅读其他语言能力的时候。虽然 C++ 标准委员会的专家也在编写一些高质量的书籍，但是书籍从翻译到出版通常需要较长时间，而且一些词语或者意义都可能在翻译中丢失。而翻译者通常也会觉得技术书籍的翻译是门苦差，很多行话、术语通常难以找到准确的中文表达方式。这么一来不同的翻译者会使用不同的术语，即使是在同一本书中，有时同一术语也会翻译成不同的中文。这些状况都是 C++ 知识传播的阻碍，会拖慢 C++11 语言被中国程序员接受的进程。

基于以上种种原因，我们决定本书让母语是中文，并且了解国内外技术差距的 IBM 上海实验室的同事编写。我们知道，在中国有非常多的 C++ 狂热爱好者正等着学习关于自己最爱的编程语言的新知识。而新的 C++11 也会招来大量的初级、中级用户，而本书也正好能满足这些人的需求。

所以，如果你属于以下几种状况之一，将会非常适合阅读本书：

- C 语言经验非常丰富且正在期待着看看 C++11 新功能的读者。
- 使用 C++98/03 并期待使用新的 C++11 的程序员。
- 任何对新的 C++11 语言感兴趣的人。

在本书中，我们引述了 C++ 之父 Bjarne Stroustrup 教授关于 C++11 的设计原则。而事实上，本书的章节划分也是基于这些设计原则的，读者在第 1 章可以找到相关信息，而剩余章节则是基于该原则对每个 C++11 语言进行的划分。对于每个特性，本书将根据其相关的论文展开描述，讲解如设计的缘由、语法规则、如何使用等内容。而书后的附录，则包括当前的

C++11 编译器支持状况、不兼容性、废弃的特性，以及论文的链接等内容。

在读完本书后，读者应该能够回答以下问题：

- ❑什么是 lambda，及怎么样使用它是最好的？
- ❑decltype 和 auto 类型推导有什么关系？
- ❑什么是移动语义，以及（右值引用）是如何解决转发问题的？
- ❑default/deleted 函数以及 override 是怎么回事？
- ❑异常描述符被什么替代了？ noexcept 是如何工作的？
- ❑什么是原子类型以及新的内存模型？
- ❑如何在 C++11 中做并行编程？

对于标准程序库，我们在本书中并没有介绍。这部分内容可能会成为我们下一本书的内容。这意味着我们将在下一本书中不仅会描述新的算法、新的类库，还会更多地描述原子类型。虽然出于性能考虑，大多数的编译器都是通过语言特性的方式来实现原子类型的，但在 C++11 标准中，原子类型却被视为一种库特性，因其可以通过库的方式来实现。同样的，这样一本书也不会教读者基础的 C++ 知识，如果读者想了解这方面的内容，我们推荐 Stroustrup 教授的《Programming principles & Practice Using C++》（中文译为：《C++ 程序设计原理与实践》，华章公司已出版）。该书是 Stroustrup 教授以其在德克萨斯 A&M 大学教授的课程为基础编写的。

对 C++11 特性感兴趣的读者可以顺序阅读本书。当然，读者也可以直接阅读自己感兴趣的章节，但是读者阅读时肯定会发现，这些特性基本和其他的特性一样，遵从了相同的设计准则。

我们也希望本书对你的职业或者个人学习起到积极的作用。当然，我们在合作写作本书，以及在为 IBM C++ 编译器开发 C++11 特性时，也颇有收获。

本书的编写经历了较长的时间，但这是值得的。我在 C++ 标准委员会工作的时候，只是在考虑写这样一本书，而官孝峰则让我从这样的考虑转到了动手行动。继而他还激励并领导其他成员共同参与，最终完成了本书。

此外，我要感谢我的一些正式的以及非正式的导师，比如 Bjarne Stroustrup、Herb Sutter、Hans Boehm、Anthony Williams、Scott Meyers，以及许多其他人，通过阅读他们的著作，或观察他们在委员会中的工作，我学会了很多。当然，更要感谢 IBM 为我们提供的平台、时间，以及各种便利，因为有了这些最终我们才能够超越自我，为新一代的程序员做一些事情，即使这样的事情可能微不足道。还要感谢的是我的家人，Sophie、Cameron、Spot（猫）和 Susan，让我能够在空闲时间完成书籍编写。

Michael

前　言

为什么要写这本书

相比其他语言的频繁更新，C++ 语言标准已经有十多年没有真正更新过了。而上一次标准制定，正是面向对象概念开始盛行的时候。较之基于过程的编程语言，基于面向对象、泛型编程等概念的 C++ 无疑是非常先进的，而 C++98 标准的制定以及各种符合标准的编译器的出现，又在客观上推动了编程方法的革命。因此在接下来的很多年中，似乎人人都在学习并使用 C++。商业公司在邀请 C++ 专家为程序员讲课，学校里老师在为学生绘声绘色地讲解面向对象编程，C++ 的书籍市场也是百花齐放，论坛、BBS 的 C++ 板块则充斥了大量各种关于 C++ 的讨论。随之而来的，招聘启事写着“要求熟悉 C++ 编程”，派生与继承成为了面试官审视毕业生基础知识的重点。凡此种种，不一而足。于是 C++ 语言“病毒性”地蔓延到各种编程环境，成为了使用最为广泛的编程语言之一。

十来年的时光转瞬飞逝，各种编程语言也在快马加鞭地向前发展。如今流行的编程语言几乎无一不支持面向对象的概念。即使是古老的语言，也通过了制定新标准，开始支持面向对象编程。随着 Web 开发、移动开发逐渐盛行，一些新流行起来的编程语言，由于在应用的快速开发、调试、部署上有着独特的优势，逐渐成为了这些新领域中的主流。不过这并不意味着 C++ 正在失去其阵地。身为 C 的“后裔”，C++ 继承了 C 能够进行底层操作的特性，因此，使用 C/C++ 编写的程序往往具有更佳的运行时性能。在构建包括操作系统的各种软件层，以及构建一些对性能要求较高的应用程序时，C/C++ 往往是最佳选择。更一般地讲，即使是由其他语言编写的程序，往往也离不开由 C/C++ 编写的编译器、运行库、操作系统，或者虚拟机等提供支持。因此，C++ 已然成为了编程技术中的中流砥柱。如果用个比喻来形容 C++，那么可以说这十来年 C++ 正是由“锋芒毕露”的青年时期走向“成熟稳重”的中年时期。

不过十来年对于编程语言来说也是个很长的时间，长时间的沉寂甚至会让有的人认为，C++ 就是这样一种语言：特性稳定，性能出色，易于学习而难于精通。长时间使用 C++ 的程序员也都熟悉了 C++ 毛孔里每一个特性，甚至是现实上的一些细微的区别，比如各种编译器对 C++ 扩展的区别，也都熟稔于心。于是这个时候，C++11 标准的横空出世，以及 C++ 之父 Bjarne Stroustrup 的一句“看起来像一门新语言”的说法，无疑让很多 C++ 程序员有些诚惶诚恐：C++11 是否又带来了编程思维的革命？C++11 是否保持了对 C++98 及 C 的兼容？旧有的 C++ 程序到了 C++11 是否需要被推倒重来？

事实上这些担心都是多余的。相比于 C++98 带来的面向对象的革命性，C++11 带来的

却并非“翻天覆地”式的改变。很多时候，程序员保持着“C++98 式”的观点来看待 C++11 代码也同样是合理的。因为在编程思想上，C++11 依然遵从了一贯的面向对象的思想，并深入加强了泛型编程的支持。从我们的观察来看，C++11 更多的是对步入“成熟稳重”的中年时期的 C++ 的一种改造。比如，像 auto 类型推导这样的新特性，展现出的是语言的亲和力；而右值引用、移动语义的特性，则着重于改变一些使用 C++ 程序库时容易发生的性能不佳的状况。当然，C++11 中也有局部的创新，比如 lambda 函数的引入，以及原子类型的设计等，都体现了语言与时俱进的活力。语言的诸多方面都在 C++11 中再次被锤炼，从而变得更加合理、更加条理清晰、更加易用。C++11 对 C++ 语言改进的每一点，都呈现出了经过长时间技术沉淀的编程语言的特色与风采。所以从这个角度上看，学习 C++11 与 C++98 在思想上是一脉相承的，程序员可以用较小的代价对 C++ 的知识进行更新换代。而在现实中，只要修改少量已有代码（甚至不修改），就可以使用 C++11 编译器对旧有代码进行升级编译而获得新标准带来的好处，这也非常具有实用性。因此，从很多方面来看，C++ 程序员都应该乐于升级换代已有的知识，而学习及使用 C++11 也正是大势所趋。

在本书开始编写的时候，C++11 标准刚刚发布一年，而本书出版的时候，C++11 也只不过才诞生了两年。这一两年，各个编译器厂商或者组织都将支持 C++11 新特性作为了一项重要工作。不过由于 C++11 的语言特性非常的多，因此本书在接近完成时，依然没有一款编译器支持 C++11 所有的新特性。但从从业者的角度看，C++11 迟早会普及，也迟早会成为 C++ 程序员的首选，因此即使现阶段编译器对 C++ 新特性的支持还不充分，但还是有必要在这个时机推出一本全面介绍 C++11 新特性的中文图书。希望通过这样的图书，使得更多的中国程序员能够最快地了解 C++11 新语言标准的方方面面，并且使用最新的 C++11 编译器来从各方面提升自己编写的 C++ 程序。

读者对象

本书针对的对象是已经学习过 C++，并想进一步学习、了解 C++11 的程序员。这里我们假定读者已经具备了基本的 C++ 编程知识，并掌握了一定的 C++ 编程技巧（对于 C++ 的初学者来说，本书阅读起来会有一定的难度）。通过本书，读者可以全面而详细地了解 C++11 对 C++ 进行的改造。无论是试图进行更加精细的面向对象程序编写，或是更加容易地进行泛型编程，或是更加轻松地改造使用程序库等，读者都会发现 C++11 提供了更好的支持。

本书作者和书籍支持

本书的作者都是编译器行业的从业者，主要来自于 IBM XL 编译器中国开发团队。IBM XL 编译器中国开发团队创立于 2010 年，拥有编译器前端、后端、性能分析、测试等各方面的人员，工作职责涵盖了 IBM XL C/C++ 及 IBM XL Fortran 编译器的开发、测试、发布等与编译器产品相关的方方面面。虽然团队成立时间不长，成员却都拥有比较丰富的编译器开发经验，对 C++11 的新特性也有较好的理解。此外，IBM 北美编译器团队成员 Michael（他是 C++ 标准委员会的成员）也参加了本书的编写工作。在书籍的编写上，Michael 为本书

拟定了提纲、确定了章节主题，并直接编写了本书的首章。其余作者则分别对 C++11 各种新特性进行了详细研究讨论，并完成了书稿其余各章的撰写工作。在书稿完成后，除了请 Michael 为本书的部分章节进行了审阅并提出修改意见外，我们又邀请了 IBM 中国信息开发部及 IBM 北京编译器团队的一些成员对本书进行了详细的审阅。虽然在书籍的策划、编写、审阅上我们群策群力，尽了最大的努力，以保证书稿质量，不过由于 C++11 标准发布时间不长，理解上的偏差在所难免，因此本书也可能在特性描述中存在一些不尽如人意或者错误的地方，希望读者、同行等一一为我们指出纠正。我们也会通过博客（http://ibm.co/HK0GCx）、微博（www.weibo.com/ibmcompiler）发布与本书相关的所有信息，并与本书读者共同讨论、进步。

如何阅读本书

读者在书籍阅读中可能会发现，本书的一些章节对 C++ 基础知识要求较高，而某些特性很可能很难应用于自己的编程实践。这样的情况应该并不少见，但这并不是这门语言缺乏亲和力，或是读者缺失了背景知识，这诚然是由于 C++ 的高成熟度导致的。在 C++11 中，不少新特性都会局限于一些应用场景，比如说库的编写，而编写库却通常不是每个程序员必须的任务。为了避免这样的状况，本书第 1 章对 C++11 的语言新特性进行了分类，因此读者可以选择按需阅读，对不想了解的部分予以略过。一些本书的使用约定，读者也可以在第 1 章中找到。

致谢

在这里我们要对 IBM 中国信息开发部的陈晶（作者之一）、卢昉、付琳，以及 IBM 北京编译器团队的冯威、许小羽、王颖对本书书稿详尽细致的审阅表示感谢，同时也对他们专业的工作素养表示由衷的钦佩。此外，我们也要感谢 IBM XL 编译器中国开发团队的舒蓓、张嗣元两位经理在本书编写过程中给予的大力支持。而 IBM 图书社区的刘慎峰及华章图书的杨福川编辑的辛勤工作则保证了本书的顺利出版，在这里我们也要对他们以及负责初审工作的孙海亮编辑说声谢谢。此外，我们还要感谢各位作者的家人在书籍编写过程中给予作者的体谅与支持。最后要感谢的是本书的读者，感谢你们对本书的支持，希望通过这本书，我们能够一起进入 C++ 编程的新时代。

IBM XL 编译器中国开发团队

目　录

免责声明
序
前言

第 1 章　新标准的诞生 …… 1
1.1　曙光：C++11 标准的诞生 …… 1
1.1.1　C++11/C++0x（以及 C11/C1x）——新标准诞生 …… 1
1.1.2　什么是 C++11/C++0x …… 2
1.1.3　新 C++ 语言的设计目标 …… 3
1.2　今时今日的 C++ …… 5
1.2.1　C++ 的江湖地位 …… 5
1.2.2　C++11 语言变化的领域 …… 5
1.3　C++11 特性的分类 …… 7
1.4　C++ 特性一览 …… 11
1.4.1　稳定性与兼容性之间的抉择 …… 11
1.4.2　更倾向于使用库而不是扩展语言来实现特性 …… 12
1.4.3　更倾向于通用的而不是特殊的手段来实现特性 …… 13
1.4.4　专家新手一概支持 …… 13
1.4.5　增强类型的安全性 …… 14
1.4.6　与硬件紧密合作 …… 14
1.4.7　开发能够改变人们思维方式的特性 …… 15
1.4.8　融入编程现实 …… 16
1.5　本书的约定 …… 17
1.5.1　关于一些术语的翻译 …… 17
1.5.2　关于代码中的注释 …… 17
1.5.3　关于本书中的代码示例与实验平台 …… 18

第 2 章　保证稳定性和兼容性 …… 19
2.1　保持与 C99 兼容 …… 19
2.1.1　预定义宏 …… 19
2.1.2　__func__ 预定义标识符 …… 20
2.1.3　_Pragma 操作符 …… 22
2.1.4　变长参数的宏定义以及 __VA_ARGS__ …… 22
2.1.5　宽窄字符串的连接 …… 23
2.2　long long 整型 …… 23
2.3　扩展的整型 …… 25
2.4　宏 __cplusplus …… 26
2.5　静态断言 …… 27
2.5.1　断言：运行时与预处理时 …… 27
2.5.2　静态断言与 static_assert …… 28
2.6　noexcept 修饰符与 noexcept 操作符 …… 32
2.7　快速初始化成员变量 …… 36
2.8　非静态成员的 sizeof …… 39
2.9　扩展的 friend 语法 …… 40
2.10　final/override 控制 …… 44
2.11　模板函数的默认模板参数 …… 48
2.12　外部模板 …… 50
2.12.1　为什么需要外部模板 …… 50
2.12.2　显式的实例化与外部模板的声明 …… 52
2.13　局部和匿名类型作模板实参 …… 54
2.14　本章小结 …… 55

第 3 章　通用为本，专用为末 …… 57
3.1　继承构造函数 …… 57

3.2 委派构造函数 ····· 62
3.3 右值引用：移动语义和完美转发 ····· 68
3.3.1 指针成员与拷贝构造 ····· 68
3.3.2 移动语义 ····· 69
3.3.3 左值、右值与右值引用 ····· 75
3.3.4 std::move：强制转化为右值 ····· 80
3.3.5 移动语义的一些其他问题 ····· 82
3.3.6 完美转发 ····· 85
3.4 显式转换操作符 ····· 89
3.5 列表初始化 ····· 92
3.5.1 初始化列表 ····· 92
3.5.2 防止类型收窄 ····· 96
3.6 POD 类型 ····· 98
3.7 非受限联合体 ····· 106
3.8 用户自定义字面量 ····· 110
3.9 内联名字空间 ····· 113
3.10 模板的别名 ····· 118
3.11 一般化的 SFINEA 规则 ····· 119
3.12 本章小结 ····· 121
第 4 章 新手易学，老兵易用 ····· 123
4.1 右尖括号 > 的改进 ····· 123
4.2 auto 类型推导 ····· 124
4.2.1 静态类型、动态类型与类型推导 ····· 124
4.2.2 auto 的优势 ····· 126
4.2.3 auto 的使用细则 ····· 130
4.3 decltype ····· 134
4.3.1 typeid 与 decltype ····· 134
4.3.2 decltype 的应用 ····· 136
4.3.3 decltype 推导四规则 ····· 140
4.3.4 cv 限制符的继承与冗余的符号 ··· 143
4.4 追踪返回类型 ····· 145
4.4.1 追踪返回类型的引入 ····· 145
4.4.2 使用追踪返回类型的函数 ····· 146
4.5 基于范围的 for 循环 ····· 150
4.6 本章小结 ····· 153
第 5 章 提高类型安全 ····· 155
5.1 强类型枚举 ····· 155
5.1.1 枚举：分门别类与数值的名字 ··· 155
5.1.2 有缺陷的枚举类型 ····· 156
5.1.3 强类型枚举以及 C++11 对原有枚举类型的扩展 ····· 160
5.2 堆内存管理：智能指针与垃圾回收 ····· 163
5.2.1 显式内存管理 ····· 163
5.2.2 C++11 的智能指针 ····· 164
5.2.3 垃圾回收的分类 ····· 167
5.2.4 C++ 与垃圾回收 ····· 169
5.2.5 C++11 与最小垃圾回收支持 ····· 170
5.2.6 垃圾回收的兼容性 ····· 172
5.3 本章小结 ····· 173
第 6 章 提高性能及操作硬件的能力 ····· 174
6.1 常量表达式 ····· 174
6.1.1 运行时常量性与编译时常量性 ··· 174
6.1.2 常量表达式函数 ····· 176
6.1.3 常量表达式值 ····· 178
6.1.4 常量表达式的其他应用 ····· 180
6.2 变长模板 ····· 183
6.2.1 变长函数和变长的模板参数 ····· 183
6.2.2 变长模板：模板参数包和函数参数包 ····· 185
6.2.3 变长模板：进阶 ····· 189
6.3 原子类型与原子操作 ····· 196
6.3.1 并行编程、多线程与 C++11 ····· 196
6.3.2 原子操作与 C++11 原子类型 ····· 197
6.3.3 内存模型，顺序一致性与 memory_order ····· 203
6.4 线程局部存储 ····· 214
6.5 快速退出：quick_exit 与 at_quick_exit ····· 216
6.6 本章小结 ····· 219

第 7 章　为改变思考方式而改变……220
7.1　指针空值——nullptr……220
7.1.1　指针空值：从 0 到 NULL，再到 nullptr……220
7.1.2　nullptr 和 nullptr_t……223
7.1.3　一些关于 nullptr 规则的讨论……225
7.2　默认函数的控制……227
7.2.1　类与默认函数……227
7.2.2　“= default”与“= deleted”……230
7.3　lambda 函数……234
7.3.1　lambda 的一些历史……234
7.3.2　C++11 中的 lambda 函数……235
7.3.3　lambda 与仿函数……238
7.3.4　lambda 的基础使用……240
7.3.5　关于 lambda 的一些问题及有趣的实验……243
7.3.6　lambda 与 STL……247
7.3.7　更多的一些关于 lambda 的讨论…254
7.4　本章小结……256
第 8 章　融入实际应用……258
8.1　对齐支持……258
8.1.1　数据对齐……258
8.1.2　C++11 的 alignof 和 alignas……261
8.2　通用属性……267
8.2.1　语言扩展到通用属性……267
8.2.2　C++11 的通用属性……268
8.2.3　预定义的通用属性……270
8.3　Unicode 支持……274
8.3.1　字符集、编码和 Unicode……274
8.3.2　C++11 中的 Unicode 支持……276
8.3.3　关于 Unicode 的库支持……280
8.4　原生字符串字面量……284
8.5　本章小结……286
附录 A　C++11 对其他标准的不兼容项目……287
附录 B　弃用的特性……294
附录 C　编译器支持……301
附录 D　相关资源……304

第①章

新标准的诞生

从最初的代号 C++0x 到最终的名称 C++11，C++ 的第二个真正意义上的标准姗姗来迟。可以想象，这个迟来的标准必定遭遇了许多的困难，而 C++ 标准委员会应对这些困难的种种策略，则构成新的 C++ 语言基因，我们可以从新的 C++11 标准中逐一体会。而客观上，这些基因也决定了 C++11 新特性的应用范畴。在本章中，我们会从设计思维和应用范畴两个维度对所有的 C++11 新特性进行分类，并依据这种分类对一些特性进行简单的介绍，从而一览 C++11 的全景。

1.1 曙光：C++11 标准的诞生

1.1.1 C++11/C++0x（以及 C11/C1x）——新标准诞生

2011 年 11 月，在印第安纳州布卢明顿市，“八月印第安纳大学会议”（August Indiana University Meeting）缓缓落下帷幕。这次会议的结束，意味着长久以来以 C++0x 为代号的 C++11 标准终于被 C++ 标准委员会批准通过。至此，C++ 新标准尘埃落定。从 C++98 标准通过的时间开始计算，C++ 标准委员会，即 WG21，已经为新标准工作了 11 年多的时间。对于一个编程语言标准而言，11 年显然是个非常长的时间。其间我们目睹了面向对象编程的盛极，也见证了泛型编程的风起云涌，还见证了 C++ 后各种新的流行编程语言的诞生。不过在新世纪第二个 10 年的伊始，C++ 的标准终于二次来袭。

事实上，在 2003 年 WG21 曾经提交了一份技术勘误表（Technical Corrigendum，简称 TC1）。这次修订使得 C++03 这个名字已经取代了 C++98 成为 C++11 之前的最新 C++ 标准名称。不过由于 TC1 主要是对 C++98 标准中的漏洞进行修复，核心语言规则部分则没有改动，因此，人们还是习惯地把两个标准合称为 C++98/03 标准。

注意 在本书中，但凡是 C++98 和 C++03 标准没有差异时，我们都会沿用 C++98/03 这样的俗称，或者直接简写为 C++98。如果涉及 TC1 中所提出的微小区别，我们会使用 C++98 和 C++03 来分别指代两种 C++ 标准。

C++11 是一种新语言的开端。虽然设计 C++11 的目的是为了要取代 C++98/03，不过相比于 C++03 标准，C++11 则带来了数量可观的变化，这包括了约 140 个新特性，以及对

C++03 标准中约 600 个缺陷的修正。因此，从这个角度看来 C++11 更像是从 C++98/03 中孕育出的一种新语言。正如当年 C++98/03 为 C++ 引入了如异常处理、模板等许多让人耳目一新的新特性一样，C++11 也通过大量新特性的引入，让 C++ 的面貌焕然一新。这些全新的特性以及相应的全新的概念，都是我们要在本书中详细描述的。

1.1.2 什么是 C++11/C++0x

C++0x 是 WG21 计划取代 C++98/03 的新标准代号。这个代号还是在 2003 年的时候取的。当时委员会乐观地估计，新标准会在 21 世纪的第一个 10 年内完成。从当时看毕竟还有 6 年的时间，确实无论如何也该好了。不过 2010 新年钟声敲响的时候，WG21 内部却还在为一些诸如哪些特性该放弃，哪些特性该被削减的议题而争论。于是所有人只好接受这个令人沮丧的事实：新标准没能准时发布。好在委员会成员保持着乐观的情绪，还常常相互开玩笑说，x 不是一个 0 到 9 的十进制数，而应该是一个十六进制数，我们还可以有 A、B、C、D、E、F。虽然这是个玩笑，但也有点认真的意思，如果需要，WG21 会再使用“额外”的 6 年，在 2015 年之前完成标准。不过众所周知的，WG21“只”再花了两年时间就完成了 C++11 标准。

注意 C 语言标准委员会（C committee）WG14 也几乎在同时开始致力于取代 C99 标准。不过相比于 WG21，WG14 对标准完成的预期更加现实。因为他们使用的代号是 C1x，这样新的 C 标准完成的最后期限将是 2019 年。事实上 WG14 并没用那么长时间，他们最终在 2011 年通过了提案，也就是 C11 标准。

从表 1-1 中可以看到 C++ 从诞生到最新通过的 C++11 标准的编年史。

表 1-1 C++ 发展编年史

日 期	事 件
1990 年	*The Annotated C++ Reference Manual*，M.A.Ellis 和 B.Stroustrup 著。主要描述了 C++ 核心语言，没有涉及库
1998 年	第一个国际化的 C++ 语言标准：IOS/IEC 15882:1998。包括了对核心语言及 STL、locale、iostream、numeric、string 等诸多特性的描述
2003 年	第二个国际化的 C++ 语言标准：IOS/IEC 15882:2003。核心语言及库与 C++98 保持了一致，但包含了 TC1（Technical Corrigendum 1, 技术勘误表 1）。自此，C++03 取代了 C++98
2005 年	TR1（Technical Report 1，技术报告 1）：IOS/IEC TR 19768:2005。核心语言不变。TR1 作为标准的非规范出版物，其包含了 14 个可能进入新标准的新程序库
2007 年 9 月	SC22 注册（特性）表决。通过了 C++0x 中核心特性
2008 年 9 月	SC22 委员会草案（Committee Draft, CD）表决。基本上所有 C++0x 的核心特性都完成了，新的 C++0x 标准草稿包括了 13 个源自 TR1 的库及 70 个库特性，修正了约 300 个库缺陷。此外，新标准草案还包括了 70 多个语言特性及约 300 个语言缺陷的修正
2010 年 3 月	SC22 最终委员会草案（Final Committee Draft, FCD）表决。所有核心特性都已经完成，处理了各国代表的评议

（续）

日　期	事　件
2011 年 11 月	JTC1 C++11 最终国际化标准草案（Final Draft International Standard, FDIS）发布，即 IOS/IEC 15882:2011。新标准在核心语言部分和标准库部分都进行了很大的改进，这包括 TR1 的大部分内容。但整体的改进还是与先前的 C++ 标准兼容的
2012 年 2 月	在 ANSI 和 ISO 商店可以以低于原定价的价格买到 C++11 标准

注意 语言标准的发布通常有两种——规范的（Normative）及不规范的（Non-normative）。前者表示内容通过了批准（ratified），因此是正式的标准，而后者则不是。不过不规范的发布通常是有积极意义的，比方说 TR1，它就是不规范的标准，但是后来很多 TR1 的内容都成为了 C++11 标准的一部分。

图 1-1 比较了两个语言标准委员会（WG21，WG14）制定新标准的工作进程，其中一些重要时间点都标注了出来。

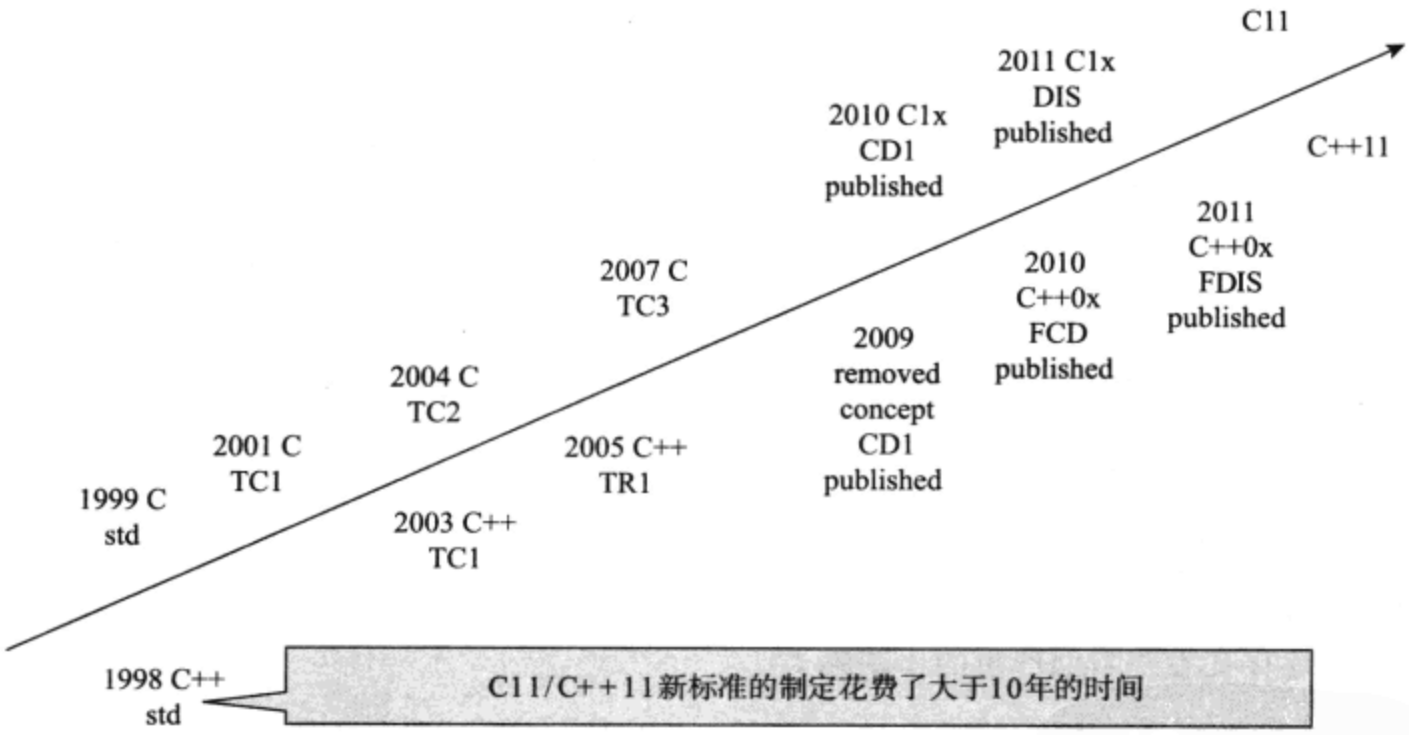


图 1-1　WG21 和 WG14 制定新语言标准的工作进程

1.1.3　新 C++ 语言的设计目标

如果读者已经学习过 C++98/03，就可以发现 C++98/03 的设计目标如下：

- 比 C 语言更适合系统编程（且与 C 语言兼容）。
- 支持数据抽象。
- 支持面向对象编程。
- 支持泛型编程。

这些特点使得面向对象编程和泛型编程在过去的 10 ~ 20 年内成为编程界的明星。不过从那时开始，C++ 的发展就不仅仅是靠学者的远见前瞻去推动的，有时也会借由一些“奇缘”而演进。比方说，C++ 模板就是这样一个“奇缘”。它使得 C++ 近乎成为了一种函数式编程语言，而且还使得 C++ 程序员拥有了模板元编程的能力。但是凡事有两面，C++98/03 中的一些较为激进的特性，比如说动态异常处理、输出模板，现在回顾起来则是不太需要的。当然，这是由于我们有了“后见之明”，或者由于这些特性在新情况下不再适用，又或者它们影响了 C++11 的新特性的设计。因此一部分这样的特性已经被 C++11 弃用了。在附录 B 中我们会一一列出这些弃用的特性，并分析其被弃用的原因。

而 C++11 的整体设计目标如下：

- 使得 C++ 成为更好的适用于系统开发及库开发的语言。
- 使得 C++ 成为更易于教学的语言（语法更加一致化和简单化）。
- 保证语言的稳定性，以及和 C++03 及 C 语言的兼容性。

我们可以分别解释一下。

首先，使 C++ 成为更好的适用于系统开发及库开发的语言，意味着标准并不只是注重为某些特定的领域提供专业化功能，比如专门为 Windows 开发提供设计，或者专门为数值计算提供设计。标准希望的是使 C++ 能够对各种系统的编程都作出贡献。

其次，使得 C++ 更易于教学，则意味着 C++11 修复了许多令程序员不安的语言“毒瘤”。这样一来，C++ 语法显得更加一致化，新手使用起来也更容易上手，而且有了更好的语法保障。其实语言复杂也有复杂的好处，比如 ROOTS、DEALII 等一些复杂科学运算的算法，它们的作者非常喜爱泛型编程带来的灵活性，于是 C++ 语言最复杂的部分正好满足了他们的需求。但是在这个世界上，新手总是远多于专家。而即使是专家，也常常只是精通自己的领域。因此语言不应该复杂到影响人们的学习。本书作者之一也是 WG21 中的一员，从结果上看，无论读者怎么看待 C++11，委员会大多数人都认同 C++11 达成了易于教学这个目标（即使其中还存在着些看似严重的小缺陷）。

最后，则是语言的稳定性。经验告诉我们，伟大的编程语言能够长期存活下来的原因还是因为语言的设计突出了实用性。事实上，在标准制定过程中，委员会承担了很多压力，这些压力源自于大家对加入更多语言特性的期盼——每一个人都希望将其他编程语言中自己喜欢的特性加入到新的 C++ 中。对于这些热烈而有些许盲目的期盼，委员会成员在 Bjarne Stroustrup 教授的引导下，选择了不断将许多无关的特性排除在外。其目的是防止 C++ 成为一个千头万绪的但功能互不关联的语言。而如同现在看到的那样，C++11 并非大而无序，相反地，许多特性可以良好地协作，进而达到“1 + 1 > 2”的效果。可以说，有了这些努力，今天的读者才能够使用稳定而强大的 C++11，而不用担心语言本身存在着混乱状况甚至是冲突。

值得一提的是，虽然在取舍新语言特性方面标准委员会曾面临过巨大压力，但与此同

时，标准委员会却没有收集到足够丰富的库的新特性。作为一种通用型语言，C++ 是否是成功，通常会依赖于不同领域中 C++ 的使用情况，比如科学计算、游戏、制造业、网络编程等。在 C++11 通过的标准库中，服务于各个领域的新特性确实还是太少了。因此很有可能在下一个版本的 C++ 标准制定中，如何标准化地使用库将成为热门话题，标准委员会也准备好了接受来自这方面的压力。

1.2 今时今日的 C++

1.2.1 C++ 的江湖地位

如今 C++ 依旧位列通用编程语言三甲，不过似乎没有以前那么流行了。事实上，编程语言排名通常非常难以衡量。比如，某位教授或学生用了 C++ 来教授课程应该被计算在内吗？在新的联合攻击战斗机（Joint Strike Fighter，JSF-35）的航空电子设备中使用了 C++ 编程应该计算在内吗？又或者 C++ 被用于一款流行的智能手机操作系统的编程中算不算呢？再或者是 C++ 被用于编写最流行的在线付费搜索引擎，或用于构建一款热门的第一人称射击游戏的引擎，或用于构建最热门的社交网络的代码库，这些都该计算在内吗？事实上，据我们所知，以上种种都使用了 C++ 编程。而且在构建致力于沟通软硬件的系统编程中，C++ 也常常是必不可少的。甚至，C++ 还常用于设计和编写编程语言。因此我们可以认为，编程语言价值的衡量标准应该包括数量、新颖性、质量，以及以上种种，都应该纳入“考核”。这样一来，结论就很明显了：C++ 无处不在。

1.2.2 C++11 语言变化的领域

如果谁说 C++11 只是对 C++ 语言做了大幅度的改进，那么他很可能就错过了 C++11 精彩的地方。事实上，读罢本书后，读者只需要看一眼代码，就可以说出代码究竟是 C++98/03 的，还是 C++11 的。C++11 为程序员创造了很多更有效、更便捷的代码编写方式，程序员可以用简短的代码来完成 C++98/03 中同样的功能，简单到你惊呼“天哪，怎么能这么简单”。从一些简单的数据统计上看，比起 C++98/03，C++11 大大缩短了代码编写量，依情况最多可以将代码缩短 30% ~ 80%。

那么 C++11 相对于 C++98/03 有哪些显著的增强呢？事实上，这包括以下几点：

- ❑通过内存模型、线程、原子操作等来支持本地并行编程（Native Concurrency）。
- ❑通过统一初始化表达式、auto、declytype、移动语义等来统一对泛型编程的支持。
- ❑通过 constexpr、POD（概念）等更好地支持系统编程。
- ❑通过内联命名空间、继承构造函数和右值引用等，以更好地支持库的构建。

表 1-2 列出了 C++11 批准通过的，且本书将要涉及的语言特性。这是一张相当长的表，而且一个个陌生的词汇足以让新手不知所措。不过现在还不是了解它们的时候。但看过这张

表，读者至少会有这样一种感觉：C++11 的确像是一门新的语言。如果我们将 C++98/03 标准中的特性和 C++11 放到一起，C++11 则像是个恐怖的“编程语言范型联盟”。利用它不仅仅可以写出面向对象语言的代码，也可以写出过程式编程语言代码、泛型编程语言代码、函数式编程语言代码、元编程编程语言代码，或者其他。多范型的支持使得 C++11 语言的“硬能力”几乎在编程语言中“无出其右”。

表 1-2 C++11 主要的新语言特性（中英文对照）

中文翻译	英文名称	备注
__cplusplus 宏	__cplusplus macro	
对齐支持	alignment support	
通用属性	general attribute	
原子操作	atomic operation	
auto 类型推导（初始化类型推导）	auto (type deduction from 18nitialize)	
C99 特性	C99	
强类型枚举	enum class (scoped and strongly typed enums)	
复制及再抛出异常	copy and rethrow exception	本书未讲解
常量表达式	constexpr	
decltype	decltype	
函数的默认模板参数	default template parameters for function	
显式默认和删除的函数（默认的控制）	defaulted and deleted functions (control of defaults)	
委托构造函数	delegating constructors	
并行动态初始化和析构	Dynamic Initialization and Destruction with Concurrency	本书未讲解
显式转换操作符	explicit conversion operators	
扩展的 friend 语法	extended friend syntax	
扩展的整型	extended integer types	
外部模板	extern templates	
一般化的 SFINAE 规则	generalized SFINAE rules	
统一的初始化语法和语义	Uniform initialization syntax and semantics	
非受限联合体	unrestricted union	
用户定义的字面量	user-defined literals	
变长模板	variadic templates	
类成员初始化	in-class member initializers	
继承构造函数	inherited constructors	
初始化列表	initializer lists	
lambda 函数	lambda	
局部类型用作模板参数	local classes as template arguments	
long long 整型	long long integers	
内存模型	memory model	
移动语义（参见右值引用）	move semantics（see rvalue references）	
内联名字空间	Inline namespace	

（续）

中文翻译	英文名称	备　注
防止类型收窄	Preventing narrowing	
指针空值	nullptr	
POD	POD（plain old data）	
基于范围的 for 语句	range-based for statement	
原生字符串字面量	raw string literals	
右值引用	rvalue reference	
静态断言	static assertions	
追踪返回类型语法	trailing return type syntax	
模板别名	template alias	
线程本地的存储	thread-local storage	
Unicode	Unicode	

而从另一个角度看，编程中程序员往往需要将实物、流程、概念等进行抽象描述。但通常情况下，程序员需要抽象出的不仅仅是对象，还有一些其他的概念，比如类型、类型的类型、算法，甚至是资源的生命周期，这些实际上都是 C++ 语言可以描述的。在 C++11 中，这些抽象概念常常被实现在库中，其使用将比在 C++98/03 中更加方便，更加好用。从这个角度上讲，C++11 则是一种所谓的“轻量级抽象编程语言”（Lightweight Abstraction Programming Language）。其好处就是程序员可以将程序设计的重点更多地放在设计、实现，以及各种抽象概念的运用上。

总的来说，灵活的静态类型、小的抽象概念、绝佳的时间与空间运行性能，以及与硬件紧密结合工作的能力都是 C++11 突出的亮点。而反观 C++98/03，其最强大的能力则可能是体现在能够构建软件基础架构，或构建资源受限及资源不受限的项目上。因此，C++11 也是 C++ 在编程语言领域上一次“泛化”与进步。

要实现表 1-2 中的各种特性，需要编译器完成大量的工作。对于大多数编译器供应商来说，只能分阶段地发布若干个编译版本，逐步支持所有特性（罗马从来就不是一天建成的，对吧）。大多数编译器已经开始了对 C++11 特性的支持。有 3 款编译器甚至从 2008 年前就开始支持 C++11 了：IBM 的 XL C/C++ 编译器从版本 10.1 开始。GNU 的 GCC 编译器从版本 4.3 开始，英特尔编译器从版本 10.1 开始。而微软则从 Visual Studio 2010 开始。最近，苹果的 clang/llvm 编译器也从 2010 年的版本 2.8 开始支持 C++11 新特性，并且急速追赶其他编译器供应商。在本书附录 C 中，读者可以找到现在情况下各种编译器对 C++11 的支持情况。

1.3　C++11 特性的分类

从设计目标上说，能够让各个特性协同工作是设计 C++11/0x 中最为关键的部分。委员会总希望通过特性协作取得整体大于个体的效果，但这也是语言设计过程中最困难的一点。

因此相比于其他的各种考虑，WG21 更专注于以下理念：

- ❑保持语言的稳定性和兼容性（Maintain stability and compatibility）。
- ❑更倾向于使用库而不是扩展语言来实现特性（Prefer libraries to language extensions）。
- ❑更倾向于通用的而不是特殊的手段来实现特性（Prefer generality to specialization）。
- ❑专家新手一概支持（Support both experts and novices）。
- ❑增强类型的安全性（Increase type safety）。
- ❑增强代码执行性能和操作硬件的能力（Improve performance and ability to work directly with hardware）。
- ❑开发能够改变人们思维方式的特性（Make only changes that change the way people think）。
- ❑融入编程现实（Fit into the real world）。

根据这些设计理念可以对新特性进行分类。在本书中，我们的核心章节（第 2 ~ 8 章）也会按照这样的方式进行划分。在可能的时候，我们也会为每个理念取个有趣一点儿的中文名字。

而从使用上，Scott Mayers 则为 C++11 创建了另外一种有效的分类方式，Mayers 根据 C++11 的使用者是类的使用者，还是库的使用者，或者特性是广泛使用的，还是库的增强的来区分各个特性。具体地，可以把特性分为以下几种：

- ❑类作者需要的（class writer，简称为“类作者”）
- ❑库作者需要的（library writer，简称为“库作者”）
- ❑所有人需要的（everyone，简称为“所有人”）
- ❑部分人需要的（everyone else，简称为“部分人”）

那么我们可以结合这种分类再来看一下可以怎样来学习所有的特性。下面我们通过设计理念和用户群对 C++11 特性进行分类，如表 1-3 所示。

由于 C++11 的新特性非常多，因此本书不准备涵盖所有内容。我们粗略地将特性划分为核心语言特性和库特性。而从 C++11 标准的章节划分来看（读者可以从网站上搜到接近于最终版本的草稿，正式的标准需要通过购买获得），本书将涉及 C++11 标准中第 1 ~ 16 章的语言特性部分（在 C++11 语言标准中，第 1 ~ 16 章涵盖了核心语言特性，第 17 ~ 30 章涉及库特性），而标准库将不在本书中描述。当然，这会导致许多灰色地带，因为如同我们提到的，我们总是倾向于使用库而不是语言扩展来实现一些特性，那么实际上，讲解语言核心特性也必然涉及库的内容。典型的，原子操作（atomics）就是这样一个例子。因此，在本书的编写中，我们只是不对标准库进行专门的讲解，而与核心内容相关的库内容，我们还是会有所描述的。

表 1-3 根据设计理念和用户群对 C++11 新特性进行划分

理　念	特性名称（中英文）	用户群
保持语言的稳定性和兼容性（Maintain stability and compatibility）	C99	部分人
	函数的默认模板参数 default template parameters for function	所有人
	扩展的 friend 语法 extended friend syntax	部分人
	扩展的整型 extended integer types	部分人
	外部模板 extern templates	部分人
	类成员的初始化 in-class member initializers	部分人
	局部类用作模板参数 local classes as template arguments	部分人
	long long 整型 long long integers	部分人
	__cplusplus	部分人
	noexcept	库作者
	override/final 控制 Override/final controls	部分人
	静态断言 static assertions	库作者
	类成员的 sizeof sizeof class data members	部分人
更倾向于通用的而不是特殊化的手段来实现特性（Prefer generality to specialization）	继承构造函数 Inherited constructor	类作者
	移动语义，完美转发，引用折叠 Move semantics, perfect forwarding, reference collapse	类作者
	委托构造函数 delegating constructors	类作者
	显式转换操作符 explicit conversion operators	库作者
	统一的初始化语法和语义，初始化列表，防止收窄 Uniform initialization syntax and semantics, initializer lists, Preventing narrowing	所有人
	非受限联合体 unrestricted unions (generalized)	部分人
	用户自定义字面量 UDL	部分人
	一般化 SFINAE 规则 generalized SFINAE rules	库作者
	内联名字空间 Inline Namespace	部分人
	PODs	部分人
	模板别名 template alias	所有人
专家新手一概支持（Support both experts and novices）	右尖括号 Right angle bracket	所有人
	auto	所有人
	基于范围的 for 语句 Ranged For	所有人
	decltype	库作者
	追踪返回类型语法（扩展的函数声明语法）Trailing return type syntax (extended function declaration syntax)	所有人
增强类型的安全性（Increase type safety）	强类型枚举 Strong enum	部分人
	unique_ptr, shared_ptr	类作者
	垃圾回收 ABI Garbage collection ABI	库作者

（续）

理　念	特性名称（中英文）	用户群
增强性能和操作硬件的能力（Improve performance and ability to work directly with hardware）	常量表达式 constexpr	类作者
	原子操作 / 内存模型 atomics/mm	所有人
	复制和再抛出异常 copying and rethrowing exceptions	所有人
	并行动态初始化和析构 Dynamic Initialization and Destruction with Concurrency	所有人
	变长模板 Variadic template	库作者
	线程本地的存储 thread-local storage	所有人
	快速退出进程 quick_exit Abandoning a process	所有人
开发能够改变人们思维方式的特性（Make only changes that change the way people think）	指针空值 nullptr	所有人
	显示默认和删除的函数（默认的控制）defaulted and deleted functions (control of defauls)	类作者
	lambdas	所有人
融入编程现实（Fit into the real world）	对齐支持 Alignments	部分人
	通用属性 Attributes [[carries dependency]] [[noreturn]]	部分人
	原生字符串字面量 raw string literals	所有人
	Unicode unicode characters	所有人

而之前我们提到过的“更倾向于使用库而不是扩展语言来实现特性”理念的部分，如果有可能，我们会在另一本书或者本书的下一个版本中来进行讲解。下面列出了属于该设计理念下的库特性：

- ❑算法增强 Algorithm improvements
- ❑容器增强 Container improvements
- ❑分配算符 Scoped allocators
- ❑std::array
- ❑std::forward_list
- ❑无序容器 Unordered containers
- ❑sts::tuple
- ❑类型特性 Type traits
- ❑std::function, std::bind
- ❑unique_ptr
- ❑shared_ptr
- ❑weak_ptr
- ❑线程 Threads
- ❑互斥 Mutex
- ❑锁 Locks

- ❑条件变量 Condition variables
- ❑时间工具 Time utilities
- ❑std::future, std::promises
- ❑std::async
- ❑随机数 Random numbers
- ❑正则表达式 regex

1.4 C++ 特性一览

接下来，我们会一窥 C++11 中的各种特性，了解它们的来历、用途、特色等。可能这部分对于还没有开始阅读正文的读者来说有些困难。如果有机会，我们建议读者在读完全书后再回到这里，这也是全书最好的总结。

1.4.1 稳定性与兼容性之间的抉择

通常在语言设计中，不破坏现有的用户代码和增加新的能力，这二者是需要同时兼顾的。就像之前的 C 一样，如今 C++ 在各种代码中、开源库中，或用户的硬盘中都拥有上亿行代码，那么当 C++ 标准委员会要改变一个关键字的意义，或者发明一个新的关键字时，原有代码就很可能发生问题。因为有些代码可能已经把要加入的这个准关键字用作了变量或函数的名字。

语言的设计者或许能够完全不考虑兼容性，但说实话这是个丑陋的做法，因为来自习惯的力量总是超乎人的想象。因此 C++11 只是在非常必要的情况下才引入新的关键字。WG21 在加入这些关键字的时候非常谨慎，至少从谷歌代码搜索（Google Code Search）的结果看来，这些关键字没有被现有的开源代码频繁地使用。不过谷歌代码搜索只会搜索开源代码，私人的或者企业的代码库（codebase）是不包含在内的。因此这些数据可能还有一定的局限性，不过至少这种方法可以避免一些问题。而 WG21 中也有很多企业代表，他们也会帮助 WG21 确定这些关键字是否会导致自己企业代码库中代码不兼容的问题。

C++11 的新关键字如下：

- ❑alignas
- ❑alignof decltype
- ❑auto（重新定义）
- ❑static_assert
- ❑using（重新定义）
- ❑noexcept
- ❑export（弃用，不过未来可能留作他用）
- ❑nullptr

- constexpr
- thread_local

这些新关键字都是相对于 C++98/03 来说的。当然，引入它们可能会破坏一些 C++98/03 代码，甚至更为糟糕的是，可能会悄悄地改变了原有 C++98/03 程序的目的。static_assert 就是这样一个例子。为了降低它与已有程序变量冲突的可能性，WG21 将这个关键字的名字设计得很长，甚至还包含了下划线，可以说命名丑得不能再丑了，不过在一些情况下，它还是会发生冲突，比如：

```
static_assert(4<=sizeof(int), "error:small ints");
```

这行代码的意图是确定编译时（不是运行时）系统的 int 整型的长度不小于 4 字节，如果小于，编译器则会报错说系统的整型太小了。在 C++11 中这是一段有效的代码，在 C++98/03 中也可能是有效的，因为程序员可能已经定义了一个名为 static_assert 的函数，以用于判断运行时的 int 整型大小是否不小于 4。显然这与 C++11 中的 static_assert 完全不同。

实际上，在 C++11 中还有两个具有特殊含义的新标识符：override、final。这两个标识符如何被编译器解释与它们所在的位置有关。如果这两个标识符出现在成员函数之后，它们的作用是标注一个成员函数是否可以被重载。不过读者实际上也可以在 C++11 代码中定义出 override、final 这样名称的变量。而在这样的情况下，它们只是标识了普通的变量名称而已。

我们主要会在第 2 章中看到相关的特性的描述。

1.4.2 更倾向于使用库而不是扩展语言来实现特性

相比于语言核心功能的稳定，库则总是能随时为程序员提供快速上手的、方便易用的新功能。库的能量是巨大的，Boost ⊖和一些公司私有的库（如 Qt、POOMA）的快速成长就说明了这一点。而且库有一个很大的优势，就是其改动不需要编译器实现新特性（只要接口保持一致即可），当然，更重要的是库可以用于支持不同领域的编程。这样一来，通常读者不需要非常精通 C++ 就能使用它们。

不过这些优点并不是被广泛认可的。狂热的语言爱好者总是觉得功能加入语言特性，由编译器实现了才是王道，而库只是第二选择。不过 WG21 的看法跟他们相反。事实上，如果可能，WG21 会尽量将一个语言特性转为库特性来实现。比较典型的如 C++11 中的线程，它被实现为库特性的一部分：std::thread，而不是一个内置的“线程类型”。同样的，C++11 中没有内置的关联数组（associative array）类型，而是将它们实现为如 std::unorder_map 这样的库。再者，C++11 也没有像其他语言一样在语言核心部分加入正则表达式功能，而是实现为

⊖ 在 C++ 的众多开源库，最为出名的应该是 Boost。Boost 是一个无限制的开源库，在设计和审阅的时候，都常常有 C++ 标准委员会的人参与。

std::regex 库。这样一来，C++ 语言可以尽量在保持较少的核心语言特性的同时，通过标准库扩大其功能。

从传统意义上讲，库可能是通过提供头文件来实现的。当然，有些时候库的提供者也会将一些实现隐藏在二进制代码库存档（archive）文件中。不过并非所有的库都是通过这样的方式提供的。事实上，库也有可能实现于编译器内部。比如 C++11 中的原子操作等许多内容，就通常不是在头文件或库存档中实现的。编译器会在内部就将原子操作实现为具体的机器指令，而无需在稍后去链接实实在在的库进行存档。而之所以将原子操作的内容放在库部分，也是为了满足将原子操作作为库实现的自由。从这个意义上讲，原子操作并非纯粹的“库”，因此也被我们选择性地纳入了本书的讲解中。

1.4.3 更倾向于通用的而不是特殊的手段来实现特性

如我们说到的，如果将无数互不相关的小特性加入 C++ 中，而且不加选择地批准通过，C++ 将成为一个令人眼花缭乱的“五金店”，不幸的是，这个五金店的产品虽然各有所长，凑在一起却是一盘散沙，缺乏战斗力。所以 WG21 更希望从中抽象出更为通用的手段而不是加入单独的特性来“练成” C++11 的“十八般武艺”。

显式类型转换操作符是一个很好的例子。在 C++98/03 中，可以用在构造函数前加上 explicit 关键字来声明构造函数为显式构造，从而防止程序员在代码中“不小心”将一些特定类型隐式地转换为用户自定义类型。不过构造函数并不是唯一会导致产生隐式类型转换的方法，在 C++98/03 中类型转换操作符也可以参与隐式转换，而程序员的意图则可能只是希望类型转换操作符在显式转换时发生。这是 C++98/03 的疏忽，不过在 C++11 中，我们已经可以做到这点了。

其他的一些新特性，比如继承构造函数、移动语义等，在本书的第 3 章中我们均会涉及。

1.4.4 专家新手一概支持

如果 C++ 只是适合专家的语言，那它就不可能是一门成功的语言。C++ 中虽然有许多专家级的特性，但这并不是必须学习的。通常程序员只需要学习一定的知识就可以使用 C++。而在 C++11 中，从易用的角度出发，修缮了很多特性，也铲除了许多带来坏声誉的“毒瘤”，比如一度被群起而攻之的“毒瘤”——双右尖括号。在 C++98/03 中，由于采用了最长匹配的解析规则（maximal munch parsing rule），编译器会在解析符号时尽可能多地“吸收”符号。这样一来，在模板解析的时候，编译器就会将原本是“模板的模板”识别为右移，并“理直气壮”地抛出一条令人绝望的错误信息：模板参数中不应该存在的右移。如今这个问题已经在 C++11 中被修正。模板参数内的两个右尖括号会终结模板参数，而不会导致编译器错误。当然从实现上讲，编译器只需要在原来报错的地方加入一些上下文的判断就可以避免

这样的错误了。比如：

```
vector<list<int>> veclist: //C++11 中有效，C++98/03 中无效
```

另一个 C++11 易于上手的例子则是统一初始化语法的引入。C++ 继承了 C 语言中所谓的“集合初始化语法”（aggregate initialization syntax，比如 a[] = {0, 1,};），而在设计类的时候，却只定义了形式单一的构造函数的初始化语法，比如 A a(0, 1)。所以在使用 C++98/03 的时候，编写模板会遇到障碍，因为模板作者无法知道模板用户会使用哪种类型来初始化模板。对于泛型编程来说，这种不一致则会导致不能总是进行泛型编程。而在 C++11 中，标准统一了变量初始化方法，所以模板作者可以总是在模板编写中采用集合初始化（初始化列表）。进一步地，集合初始化对于类型收窄还有一定的限制。而类型收窄也是许多让人深夜工作的奇特错误的源头。因此在 C++11 中使用了初始化列表，就等同于拥有了防止收窄和泛型编程的双重好处。

读者可以在第 4 章看到 C++11 是如何增进语言对新手的支持的。

1.4.5 增强类型的安全性

绝对的类型安全对编程语言来说几乎是不可能达到的，不过在编译时期捕捉更多的错误则是非常有益的。在 C++98/03 中，枚举类会退化为整型，因此常会与其他的枚举类型混淆。这个类型的不安全根源还是在于兼容 C 语言。在 C 中枚举用起来非常便利，在 C++ 中却是类型系统的一个大“漏勺”。因此在 C++11 中，标准引入了新的“强类型枚举”来解决这个问题。

```
enum class Color { red, blue, green };
int x = Color::red;     //C++98/03 中允许，C++11 中错误：不存在 Color->int 的转换
Color y = 7;     //C++98/03 中，C++11 中错误：不存在 int->Color conversion 的转换
Color z = red;     //C++98/03 中允许，C++11 中错误：red 不在作用域内
Color c = Color::red;     //C++98/03 中错误，C++11 中允许
```

在第 5 章中，我们会详细讲解诸如此类能够增强类型安全的 C++11 特性。

1.4.6 与硬件紧密合作

在 C++ 编程中，嵌入式编程是一个非常重要的领域。虽然一些方方圆圆的智能设备外表光鲜亮丽，但是植根于其中的技术基础也常常会是 C++。在 C++11 中，常量表达式以及原子操作都是可以用于支持嵌入式编程的重要特性。这些特性对于提高性能、降低存储空间都大有好处，比如 ROM。

C++98/03 中也具备 const 类型，不过它对只读内存（ROM）支持得不够好。这是因为在 C++ 中 const 类型只在初始化后才意味着它的值应该是常量表达式，从而在运行时不能被改变。不过由于初始化依旧是动态的，这对 ROM 设备来说并不适用。这就要求在动态初始化前就将常量计算出来。为此标准增加了 constexpr，它让函数和变量可以被编译时的常量取

代，而从效果上说，函数和变量在固定内存设备中要求的空间变得更少，因而对于手持、桌面等用于各种移动控制的小型嵌入式设备（甚至心率调整器）的 ROM 而言，C++11 也支持得更好。

在 C++11，我们甚至拥有了直接操作硬件的方法。这里指的是 C++11 中引入的原子类型。C++11 通过引入内存模型，为开发者和系统建立了一个高效的同步机制。作为开发者，通常需要保证线程程序能够正确同步，在程序中不会产生竞争。而相对地，系统（可能是编译器、内存系统，或是缓存一致性机制）则会保证程序员编写的程序（使用原子类型）不会引入数据竞争。而且为了同步，系统会自行禁止某些优化，又保证其他的一些优化有效。除非编写非常底层的并行程序，否则系统的优化对程序员来讲，基本上是透明的。这可能是 C++11 中最大、最华丽的进步。而就算程序员不乐意使用原子类型，而要使用线程，那么使用标准的互斥变量 mutex 来进行临界区的加锁和开锁也就够了。而如果读者还想要疯狂地挖掘并行的速度，或试图完全操控底层，或想找点麻烦，那么无锁（lock-free）的原子类型也可以满足你的各种“野心”。内存模型的机制会保证你不会犯错。只有在使用与系统内存单位不同的位域的时候，内存模型才无法成功地保证同步。比如说下面这个位域的例子，这样的位域常常会引发竞争（跨了一个内存单元），因为这破坏了内存模型的假定，编译器不能保证这是没有竞争的。

```
struct {int a:9; int b:7;}
```

不过如果使用下面的字符位域则不会引发竞争，因为字符位域可以被视为是独立内存位置。而在 C++98/03 中，多线程程序中该写法却通常会引发竞争。这是因为编译器可能将 a 和 b 连续存放，那么对 b 进行赋值（互斥地）的时候就有可能在 a 没有被上锁的情况下一起写掉了。原因是在单线程情况下常被视为普通的安全的优化，却没有考虑到多线程情况下的复杂性。C++11 则在这方面做出了较好的修正。

```
struct {char a; char b;}
```

与硬件紧密合作的能力使得 C++ 可以在任何系统编程中继续保持领先的位置，比如说构建设备驱动或操作系统内核，同时在一些像金融、游戏这样需要高性能后台守护进程的应用中，C++ 的参与也会大大提升其性能。

我们会在第 6 章看到相关特性的描述。

1.4.7 开发能够改变人们思维方式的特性

C++11 中一个小小的 lambda 特性是如何撬动编程世界的呢？从一方面讲，lambda 只是对 C++98/03 中带有 operator() 的局部仿函数（函数对象）包装后的“语法甜点”。事实上，在 C++11 中 lambda 也被处理为匿名的仿函数。当创建 lambda 函数的时候，编译器内部会生成这样一个仿函数，并从其父作用域中取得参数传递给 lambda 函数。不过，真正会改变人们思维方式的是，lambda 是一个局部函数，这在 C++98/03 中我们只能模仿实现该特性。

此外，当程序员开始越来越多地使用 C++11 中先进的并行编程特性时，lambda 会成为一个非常重要的语法。程序员将会发现到处都是奇怪的“lambda 笑脸”，即 ;}[⊖]，而且程序员也必须习惯在各种上下文中阅读翻译 lambda 函数。顺带一提，lambda 笑脸常会出现在每一个 lambda 表达式的终结部分。

另一个人们会改变思维方式的地方则是如何让一个成员函数变得无效。在 C++98/03 中，我们惯用的方法是将成员函数声明为私有的。如果读者不知道这种方法的用意，很可能在阅读代码的时候产生困惑。不过今天的读者非常幸运，因为在 C++11 中不再需要这样的手段。在 C++11 中我们可以通过显式默认和删除的特性，清楚明白地将成员函数设为删除的。这无疑改变了程序员编写和阅读代码的方式，当然，思考问题的方式也就更加直截了当了。

我们会在第 7 章中看到相关特性的描述。

1.4.8 融入编程现实

现实世界中的编程往往都有特殊的需求。比如在访问因特网的时候我们常常需要输入 URL，而 URL 通常都包含了斜线“/”。要在 C++ 中输入斜线却不是件容易的事，通常我们需要转义字符“\”的配合，否则斜线则可能被误认为是除法符号。所以如果读者在写网络地址或目录路径的时候，代码最终看起来就是一堆倒胃口的反斜线的组合，而且会让内容变得晦涩。而 C++11 中的原生字符串常量则可免除“转义”的需要，也可以帮助程序员清晰地呈现网络地址或文件系统目录的真实内容。

另一方面，如今 GNU 的属性（attribute）几乎无所不在，所有的编译器都在尝试支持它，以用于修饰类型、变量和函数等。不过 __attribute__((attribute-name)) 这样的写法，除了不怎么好看外，每一个编译器可能还都有它自己的变体，比如微软的属性就是以 __declspec 打头的。因此在 C++11 中，我们看到了通用属性的出现。

不过 C++11 引入通用属性更大的原因在于，属性可以在不引入额外的关键字的情况下，为编译提供额外的信息。因此，一些可以实现为关键字的特性，也可以用属性来实现（在某些情况下，属性甚至还可以在代码中放入程序供应商的名字，不过这样做存在一些争议）。这在使用关键字还是将特性实现为一个通用属性间就会存在权衡。不过最后标准委员会认为，在现在的情况下，在 C++11 中的通用属性不能破坏已有的类型系统，也不应该在代码中引起语义的歧义。也就是说，有属性的和没有属性的代码在编译时行为是一致的。所以 C++11 标准最终选择创建很少的几个通用属性——noreturn 和 carrier_dependency（其实 final、override 也一度是热门“人选”）。

属性的真正强大之处在于它们能够让编译器供应商创建他们自己的语言扩展，同时不

⊖ lambda 笑脸是一种编写 lambda 函数的编程风格，即在 lambda 函数结束时将分号与括号连写，看起来就是一个 ;} 形式的笑脸。而实际在本书第 7 章中没有采用 lambda 笑脸的编程风格。

会干扰语言或等待特性的标准化。它们可以被用于在一些域内加入特定的“方言”，甚至是在不用 pragma 语法的情况下扩展专有的并行机制（如果读者了解 OpenMP，对此会有一些体会）。

我们将在第 8 章中看到相关的描述。

1.5 本书的约定

1.5.1 关于一些术语的翻译

在 C++11 标准中，我们会涉及很多已有的或新建的术语。在本书中，这些术语我们会尽量翻译，但不求过度翻译。

在已有翻译且翻译意义已经被广为接受的情况下，我们会使用已有的翻译词汇。比如说将 class 翻译为“类”，或者将 template 翻译为“模板”。这样翻译已经为中文读者广为接受，本书则会沿用这样的译法。

而已有翻译但是意义并没有被广为接受的情况下，本书中则会考虑保留英文原文。比如说将“URL”翻译为“统一资源定址器”在我们看来就是一种典型的不良情况。通常将这样的术语翻译为中文会阻碍读者的理解。而大多数能够阅读本书的读者也会具有基本的英文阅读能力和一些常识性的计算机知识，因此本书将保留原文，以期望能够帮助读者更好地理解涉及术语的部分。

对于还没有广泛被认同的中文翻译的术语，我们会采用审慎的态度。一些时候，如果英文确实有利于理解，我们会尝试以注释的方式提供一个中文的解释，而在文中保持英文。如果翻译成中文非常利于理解，则会提供一个中文的翻译，在注释中留下英文。

1.5.2 关于代码中的注释

在本书中，如果可能我们会将一些形如 cout、printf 打印至标准输出 / 错误的内容放在代码的注释中，从读书的经验来看，我们认为这样是最方便阅读的。比如：

```
int a = 2012;
cout << "hello, world" << endl;     // hello, world
cout << a << " is doomed" << endl; // 2012 is doomed
```

同时，一些关键的、有助于读者理解代码的解释也会放在注释中。在通常情况下，注释中有了打印结果的语句不会再有其他的代码解释。如果有，我们将会以逗号将其分开。比如：

```
cout << "hello world" << endl;     // hello world, 打印 "hello world"
```

1.5.3 关于本书中的代码示例与实验平台

在本书的编写中，我们一共使用了 3 种编译器对代码进行编译，即 IBM 的 xlC++、GNU 的 g++，以及 llvm 的 clang++。我们使用的这 3 种编译器都是开发中的版本，其中 xlC++ 使用的是开发中的版本 13，g++ 使用的是开发中的版本 4.8，而 clang++ 则使用的是开发中的版本 3.2。

本书的代码大多数由作者原创，少量使用了 C++11 标准提案中的案例，以及一些网上资源。由于本书编写时，还没有编译器提供对 C++11 所有特性的完整支持，所以通常我们都会将使用的编译器、编译时采用的编译选项罗列在代码处。在本书的代码中，我们会以 g++ 编译为主，但这并不意味着其他编译器无法编译通过这些代码示例。从我们现在看到的结果而言，使用相同特性的代码，编译器的支持往往不存在很大的个体差别（这也是设立标准的意义所在）。而具体的编译器支持，读者则可以通过附录 C 获得相关的信息。

我们的代码运行平台之一是一台运行在 IBM Power 服务器上的 SUSE Linux Enterprise Server 11 (x86_64) 的虚拟机（从我们的实验看来，在该虚拟机上并没有出现与实体机器不一致之处，而不同的 Linux 也不会对我们的实验产生影响）。运行平台之二则是一台运行于 SUSE Linux Enterprise Server 10 SP2 (ppc) 的 IBM Power5+ 服务器。

第②章

保证稳定性和兼容性

作为 C 语言的嫡亲，C++ 有一个众所周知的特性——对 C 语言的高度兼容。这样的兼容性不仅体现在程序员可以较为容易地将 C 代码“升级”为 C++ 代码上，也体现在 C 代码可以被 C++ 的编译器所编译上。新的 C++11 标准也并不例外。在 C++11 中，设计者总是保证在不破坏原有设计的情况下，增加新的特性，以充分保证语言的稳定性与兼容性。本章中的新特性基本上都遵循了该设计思想。

2.1 保持与 C99 兼容

☞类别：部分人

在 C11 之前最新的 C 标准是 1999 年制定的 C99 标准。而第一个 C++ 语言标准却出现在 1998 年（通常被称为 C++98），随后的 C++03 标准也只对 C++98 进行了小的修正。这样一来，虽然 C 语言发展中的大多数改进都被引入了 C++ 语言标准中，但还是存在着一些属于 C99 标准的“漏网之鱼”。所以 C++11 将对以下 C99 特性的支持也都纳入了新标准中：

- ❑C99 中的预定义宏
- ❑__func__ 预定义标识符
- ❑_Pragma 操作符
- ❑不定参数宏定义以及 __VA_ARGS__
- ❑宽窄字符串连接

这些特性并不像语法规则一样常用，并且有的 C++ 编译器实现也都先于标准地将这些特性实现，因此可能大多数程序员没有发现这些不兼容。但将这些 C99 的特性在 C++11 中标准化无疑可以更广泛地保证两者的兼容性。我们来分别看一下。

2.1.1 预定义宏

除去语法规范等，包括标准库的接口函数定义、相关的类型、宏、常量等也都会被发布在语言标准中。相较于 C89 标准，C99 语言标准增加一些预定义宏。C++11 同样增加了对这些宏的支持。我们可以看一下表 2-1。

表 2-1 C++11 中与 C99 兼容的宏

宏 名 称	功 能 描 述
__STDC_HOSTED__	如果编译器的目标系统环境中包含完整的标准 C 库，那么这个宏就定义为 1，否则宏的值为 0
__STDC__	C 编译器通常用这个宏的值来表示编译器的实现是否和 C 标准一致。C++11 标准中这个宏是否定义以及定成什么值由编译器来决定
__STDC_VERSION__	C 编译器通常用这个宏来表示所支持的 C 标准的版本，比如 1999mmL。C++11 标准中这个宏是否定义以及定成什么值将由编译器来决定
__STDC_ISO_10646__	这个宏通常定义为一个 yyyymmL 格式的整数常量，例如 199712L，用来表示 C++ 编译环境符合某个版本的 ISO/IEC 10646 标准

使用这些宏，我们可以查验机器环境对 C 标准和 C 库的支持状况，如代码清单 2-1 所示。

代码清单 2-1

```
#include <iostream>
using namespace std;

int main() {
    cout << "Standard Clib: " << __STDC_HOSTED__ << endl;    // Standard Clib: 1
    cout << "Standard C: " << __STDC__ << endl;              // Standard C: 1
    // cout << "C Stardard version: " << __STDC_VERSION__ << endl;
    cout << "ISO/IEC " << __STDC_ISO_10646__ << endl;        // ISO/IEC 200009
}

// 编译选项 :g++ -std=c++11 2-1-1.cpp
```

在我们的实验机上，__STDC_VERSION__ 这个宏没有定义（也是符合标准规定的，如表 2-1 所示），其余的宏都可以打印出一些常量值。

预定义宏对于多目标平台代码的编写通常具有重大意义。通过以上的宏，程序员通过使用 #ifdef/#endif 等预处理指令，就可使得平台相关代码只在适合于当前平台的代码上编译，从而在同一套代码中完成对多平台的支持。从这个意义上讲，平台信息相关的宏越丰富，代码的多平台支持越准确。

不过值得注意的是，与所有预定义宏相同的，如果用户重定义（#define）或 #undef 了预定义的宏，那么后果是“未定义”的。因此在代码编写中，程序员应该注意避免自定义宏与预定义宏同名的情况。

2.1.2 __func__ 预定义标识符

很多现实的编译器都支持 C99 标准中的 __func__ 预定义标识符功能，其基本功能就是

返回所在函数的名字。我们可以看看下面这个例子，如代码清单 2-2 所示。

代码清单 2-2

```
#include <string>
#include <iostream>
using namespace std;
const char* hello() { return __func__; }
const char* world() { return __func__; }

int main(){
    cout << hello() << ", " << world() << endl; // hello, world
}

// 编译选项 :g++ -std=c++11 2-1-2.cpp
```

在代码清单 2-2 中，我们定义了两个函数 hello 和 world。利用 __func__ 预定义标识符，我们返回了函数的名字，并将其打印出来。事实上，按照标准定义，编译器会隐式地在函数的定义之后定义 __func__ 标识符。比如上述例子中的 hello 函数，其实际的定义等同于如下代码：

```
const char* hello() {
    static const char* __func__ = "hello";
    return __func__;
}
```

__func__ 预定义标识符对于轻量级的调试代码具有十分重要的作用。而在 C++11 中，标准甚至允许其使用在类或者结构体中。我们可以看看下面这个例子，如代码清单 2-3 所示。

代码清单 2-3

```
#include <iostream>
using namespace std;

struct TestStruct {
    TestStruct () : name(__func__) {}
    const char *name;
};

int main() {
    TestStruct ts;
    cout << ts.name << endl;     // TestStruct
}
// 编译选项 :g++ -std=c++11 2-1-3.cpp
```

从代码清单 2-3 可以看到，在结构体的构造函数中，初始化成员列表使用 __func__ 预定义标识符是可行的，其效果跟在函数中使用一样。不过将 __fun__ 标识符作为函数参数的默

认值是不允许的，如下例所示：

```
void FuncFail( string func_name = __func__) {};// 无法通过编译
```

这是由于在参数声明时，__func__ 还未被定义。

2.1.3 _Pragma 操作符

在 C/C++ 标准中，#pragma 是一条预处理的指令（preprocessor directive）。简单地说，#pragma 是用来向编译器传达语言标准以外的一些信息。举个简单的例子，如果我们在代码的头文件中定义了以下语句：

```
#pragma once
```

那么该指令会指示编译器（如果编译器支持），该头文件应该只被编译一次。这与使用如下代码来定义头文件所达到的效果是一样的。

```
#ifndef THIS_HEADER
#define THIS_HEADER
// 一些头文件的定义
#endif
```

在 C++11 中，标准定义了与预处理指令 #pragma 功能相同的操作符 _Pragma。_Pragma 操作符的格式如下所示：

```
_Pragma （字符串字面量）
```

其使用方法跟 sizeof 等操作符一样，将字符串字面量作为参数写在括号内即可。那么要达到与上例 #pragma 类似的效果，则只需要如下代码即可。

```
_Pragma("once");
```

而相比预处理指令 #pragma，由于 _Pragma 是一个操作符，因此可以用在一些宏中。我们可以看看下面这个例子：

```
#define CONCAT(x) PRAGMA(concat on #x)
#define PRAGMA(x) _Pragma(#x)
CONCAT( ..\concat.dir )
```

这里，CONCAT(..\concat.dir) 最终会产生 _Pragma(concat on "..\concat.dir") 这样的效果（这里只是显示语法效果，应该没有编译器支持这样的 _Pragma 语法）。而 #pragma 则不能在宏中展开，因此从灵活性上来讲，C++11 的 _Pragma 具有更大的灵活性。

2.1.4 变长参数的宏定义以及 __VA_ARGS__

在 C99 标准中，程序员可以使用变长参数的宏定义。变长参数的宏定义是指在宏定义中参数列表的最后一个参数为省略号，而预定义宏 __VA_ARGS__ 则可以在宏定义的实现部分

替换省略号所代表的字符串。比如：

```
#define PR(...) printf(__VA_ARGS__)
```

就可以定义一个 printf 的别名 PR。事实上，变长参数宏与 printf 是一对好搭档。我们可以看如代码清单 2-4 所示的一个简单的变长参数宏的应用。

代码清单 2-4

```
#include <stdio.h>

#define LOG(...)  {\
    fprintf(stderr,"%s: Line %d:\t", __FILE__, __LINE__);\
    fprintf(stderr, __VA_ARGS__);\
    fprintf(stderr,"\n");\
}

int main() {
    int x = 3;
    // 一些代码 ...
    LOG("x = %d", x); // 2-1-5.cpp: Line 12:     x = 3
}
// 编译选项 :g++ -std=c++11 2-1-5.cpp
```

在代码清单 2-4 中，定义 LOG 宏用于记录代码位置中一些信息。程序员可以根据 stderr 产生的日志追溯到代码中产生这些记录的位置。引入这样的特性，对于轻量级调试，简单的错误输出都是具有积极意义的。

2.1.5 宽窄字符串的连接

在之前的 C++ 标准中，将窄字符串（char）转换成宽字符串（wchar_t）是未定义的行为。而在 C++11 标准中，在将窄字符串和宽字符串进行连接时，支持 C++11 标准的编译器会将窄字符串转换成宽字符串，然后再与宽字符串进行连接。

事实上，在 C++11 中，我们还定义了更多种类的字符串类型（主要是为了更好地支持 Unicode），更多详细的内容，读者可以参见 8.3 与 8.4 节。

2.2 long long 整型

☞类别：部分人

相比于 C++98 标准，C++11 整型的最大改变就是多了 long long。但事实上，long long 整型本来就离 C++ 标准很近，早在 1995 年，long long 就被提议写入 C++98 标准，却被 C++ 标准委员会拒绝了。而后来，long long 类型却进入了 C99 标准，而且也事实上也被很多编译器支持。于是辗转地，C++ 标准委员会又掉头决定将 long long 纳入 C++11 标准。

long long 整型有两种：long long 和 unsigned long long。在 C++11 中，标准要求 long long 整型可以在不同平台上有不同的长度，但至少有 64 位。我们在写常数字面量时，可以使用 LL 后缀（或是 ll）标识一个 long long 类型的字面量，而 ULL（或 ull、Ull、uLL）表示一个 unsigned long long 类型的字面量。比如：

```
long long int lli = -9000000000000000000LL;
unsigned long long int ulli = -9000000000000000000ULL;
```

就定义了一个有符号的 long long 变量 lli 和无符号的 unsigned long long 变量 ulli。事实上，在 C++11 中，还有很多与 long long 等价的类型。比如对于有符号的，下面的类型是等价的：long long、signed long long、long long int、signed long long int；而 unsigned long long 和 unsigned long long int 也是等价的。

同其他的整型一样，要了解平台上 long long 大小的方法就是查看 <climits>（或 <limits.h> 中的宏）。与 long long 整型相关的一共有 3 个：LLONG_MIN、LLONG_MAX 和 ULLONG_MIN，它们分别代表了平台上最小的 long long 值、最大的 long long 值，以及最大的 unsigned long long 值。可以看看下面这个例子，如代码清单 2-5 所示。

代码清单 2-5

```
#include <climits>
#include <cstdio>
using namespace std;

int main() {
    long long ll_min = LLONG_MIN;
    long long ll_max = LLONG_MAX;
    unsigned long long ull_max = ULLONG_MAX;

    printf("min of long long: %lld\n", ll_min); // min of long long:
    -9223372036854775808
    printf("max of long long: %lld\n", ll_max); // max of long long:
    9223372036854775807
    printf("max of unsigned long long: %llu\n", ull_max);    // max of unsigned
    long long: 18446744073709551615
}
// 编译选项 :g++ -std=c++11 2-2-1.cpp
```

在代码清单 2-5 中，将以上 3 个宏打印了出来，对于 printf 函数来说，输出有符号的 long long 类型变量可以用符号 %lld，而无符号的 unsigned long long 则可以采用 %llu。18446744073709551615 用 16 进制表示是 0xFFFFFFFFFFFFFFFF（16 个 F），可知在我们的实验机上，long long 是一个 64 位的类型。

2.3 扩展的整型

☞ **类别：部分人**

程序员常会在代码中发现一些整型的名字，比如 UINT、__int16、u64、int64_t，等等。这些类型有的源自编译器的自行扩展，有的则是来自某些编程环境（比如工作在 Linux 内核代码中），不一而足。而事实上，在 C++11 中一共只定义了以下 5 种标准的有符号整型：

- ❑signed char
- ❑short int
- ❑int
- ❑long int
- ❑long long int

标准同时规定，每一种有符号整型都有一种对应的无符号整数版本，且有符号整型与其对应的无符号整型具有相同的存储空间大小。比如与 signed int 对应的无符号版本的整型是 unsigned int。

在实际的编程中，由于这 5 种基本的整型适用性有限，所以有时编译器出于需要，也会自行扩展一些整型。在 C++11 中，标准对这样的扩展做出了一些规定。具体地讲，除了标准整型（standard integer type）之外，C++11 标准允许编译器扩展自有的所谓扩展整型（extended integer type）。这些扩展整型的长度（占用内存的位数）可以比最长的标准整型（long long int，通常是一个 64 位长度的数据）还长，也可以介于两个标准整数的位数之间。比如在 128 位的架构上，编译器可以定义一个扩展整型来对应 128 位的的整数；而在一些嵌入式平台上，也可能需要扩展出 48 位的整型；不过 C++11 标准并没有对扩展出的类型的名称有任何的规定或建议，只是对扩展整型的使用规则做出了一定的限制。

简单地说，C++11 规定，扩展的整型必须和标准类型一样，有符号类型和无符号类型占用同样大小的内存空间。而由于 C/C++ 是一种弱类型语言[⊖]，当运算、传参等类型不匹配的时候，整型间会发生隐式的转换，这种过程通常被称为整型的提升（Integral promotion）。比如如下表达式：

```
(int) a + (long long)b
```

通常就会导致变量 (int)a 被提升为 long long 类型后才与 (long long)b 进行运算。而无论是扩展的整型还是标准的整型，其转化的规则会由它们的“等级”（rank）决定。而通常情况，我们认为有如下原则：

- ❑长度越大的整型等级越高，比如 long long int 的等级会高于 int。

⊖ 关于 C/C++ 是强类型语言还是弱类型语言存在一些争议，请参见 http://stackoverflow.com/questions/430182/is-c-strongly-typed。

- 长度相同的情况下，标准整型的等级高于扩展类型，比如 long long int 和 _int64 如果都是 64 位长度，则 long long int 类型的等级更高。
- 相同大小的有符号类型和无符号类型的等级相同，long long int 和 unsigned long long int 的等级就相同。

而在进行隐式的整型转换的时候，一般是按照低等级整型转换为高等级整型，有符号的转换为无符号。这种规则其实跟 C++98 的整型转换规则是一致的。

在这样的规则支持下，如果编译器定义一些自有的整型，即使这样自定义的整型由于名称并没有被标准收入，因而可移植性并不能得到保证，但至少编译器开发者和程序员不用担心自定义的扩展整型与标准整型间在使用规则上（尤其是整型提升）存在着不同的认识了。

比如在一个 128 位的构架上，编译器可以定义 _int128_t 为 128 位的有符号整型（对应的无符号类型为 _uint128_t）。于是程序员可以使用 _int128_t 类型保存形如 +92233720368547758070 的超长整数（长于 64 位的自然数）。而不用查看编译器文档我们也会知道，一旦遇到整型提升，按照上面的规则，比如 _int128_t a，与任何短于它的类型的数据 b 进行运算（比如加法）时，都会导致 b 被隐式地转换为 _int128_t 的整型，因为扩展的整型必须遵守 C++11 的规范。

2.4 宏 __cplusplus

☞ 类别：部分人

在 C 与 C++ 混合编写的代码中，我们常常会在头文件里看到如下的声明：

```
#ifdef __cplusplus
extern "C" {
#endif
// 一些代码
#ifdef __cplusplus
}
#endif
```

这种类型的头文件可以被 #include 到 C 文件中进行编译，也可以被 #include 到 C++ 文件中进行编译。由于 extern "C" 可以抑制 C++ 对函数名、变量名等符号（symbol）进行名称重整（name mangling），因此编译出的 C 目标文件和 C++ 目标文件中的变量、函数名称等符号都是相同的（否则不相同），链接器可以可靠地对两种类型的目标文件进行链接。这样该做法成为了 C 与 C++ 混用头文件的典型做法。

鉴于以上的做法，程序员可能认为 __cplusplus 这个宏只有“被定义了”和“未定义”两种状态。事实上却并非如此，__cplusplus 这个宏通常被定义为一个整型值。而且随着标准变化，__cplusplus 宏一般会是一个比以往标准中更大的值。比如在 C++03 标准中，__cplusplus

的值被预定为 199711L，而在 C++11 标准中，宏 __cplusplus 被预定义为 201103L。这点变化可以为代码所用。比如程序员在想确定代码是使用支持 C++11 编译器进行编译时，那么可以按下面的方法进行检测：

```
#if __cplusplus < 201103L
    #error "should use C++11 implementation"
#endif
```

这里，使用了预处理指令 #error，这使得不支持 C++11 的代码编译立即报错并终止编译。读者可以使用 C++98 编译器和 C++11 的编译器分别实验一下其效果。

2.5 静态断言

☞ **类别：库作者**

2.5.1 断言：运行时与预处理时

断言（assertion）是一种编程中常用的手段。在通常情况下，断言就是将一个返回值总是需要为真的判别式放在语句中，用于排除在设计的逻辑上不应该产生的情况。比如一个函数总需要输入在一定的范围内的参数，那么程序员就可以对该参数使用断言，以迫使在该参数发生异常的时候程序退出，从而避免程序陷入逻辑的混乱。

从一些意义上讲，断言并不是正常程序所必需的，不过对于程序调试来说，通常断言能够帮助程序开发者快速定位那些违反了某些前提条件的程序错误。在 C++ 中，标准在 <cassert> 或 <assert.h> 头文件中为程序员提供了 assert 宏，用于在运行时进行断言。我们可以看看下面这个例子，如代码清单 2-6 所示。

代码清单 2-6

```
#include <cassert>
using namespace std;
// 一个简单的堆内存数组分配函数
char * ArrayAlloc(int n) {
    assert(n > 0);  // 断言，n 必须大于 0
    return new char [n];
}

int main (){
    char* a = ArrayAlloc(0);
}
// 编译选项 :g++ 2-5-1.cpp
```

在代码清单 2-6 中，我们定义了一个 ArrayAlloc 函数，该函数的唯一功能就是在堆上分配字节长度为 n 的数组并返回。为了避免意外发生，函数 ArrayAlloc 对参数 n 进行了断言，

要求其大于0。而main函数中对ArrayAlloc的使用却没有满足这个条件，那么在运行时，我们可以看到如下结果：

```
a.out: 2-5-1.cpp:6: char* ArrayAlloc(int): Assertion `n > 0' failed.
Aborted
```

在C++中，程序员也可以定义宏NDEBUG来禁用assert宏。这对发布程序来说还是必要的。因为程序用户对程序退出总是敏感的，而且部分的程序错误也未必会导致程序全部功能失效。那么通过定义NDEBUG宏发布程序就可以尽量避免程序退出的状况。而当程序有问题时，通过没有定义宏NDEBUG的版本，程序员则可以比较容易地找到出问题的位置。事实上，assert宏在<cassert>中的实现方式类似于下列形式：

```
#ifdef  NDEBUG
# define assert(expr)            (static_cast<void> (0))
#else
...
#endif
```

可以看到，一旦定义了NDBUG宏，assert宏将被展开为一条无意义的C语句（通常会被编译器优化掉）。

在2.4节中，我们还看到了#error这样的预处理指令，而事实上，通过预处理指令#if和#error的配合，也可以让程序员在预处理阶段进行断言。这样的用法也是极为常见的，比如GNU的cmathcalls.h头文件中（在我们实验机上，该文件位于/usr/include/bits/cmathcalls.h），我们会看到如下代码：

```
#ifndef _COMPLEX_H
#error "Never use <bits/cmathcalls.h> directly; include <complex.h> instead."
#endif
```

如果程序员直接包含头文件<bits/cmathcalls.h>并进行编译，就会引发错误。#error指令会将后面的语句输出，从而提醒用户不要直接使用这个头文件，而应该包含头文件<complex.h>。这样一来，通过预处理时的断言，库发布者就可以避免一些头文件的引用问题。

2.5.2 静态断言与static_assert

通过2.5.1节的例子可以看到，断言assert宏只有在程序运行时才能起作用。而#error只在编译器预处理时才能起作用。有的时候，我们希望在编译时能做一些断言。比如下面这个例子，如代码清单2-7所示。

代码清单2-7

```
#include <cassert>
using namespace std;
```

```
// 枚举编译器对各种特性的支持，每个枚举值占一位
enum FeatureSupports {
    C99          =   0x0001,
    ExtInt       =   0x0002,
    SAssert      =   0x0004,
    NoExcept     =   0x0008,
    SMAX         =   0x0010,
};

// 一个编译器类型，包括名称、特性支持等
struct Compiler{
    const char * name;
    int spp;     // 使用 FeatureSupports 枚举
};

int main() {
    // 检查枚举值是否完备
    assert((SMAX - 1) == (C99 | ExtInt | SAssert | NoExcept));

    Compiler a = {"abc", (C99 | SAssert)};
    // ...
    if (a.spp & C99) {
        // 一些代码 ...
    }
}
// 编译选项 :g++ 2-5-2.cpp
```

代码清单 2-7 所示的是 C 代码中常见的“按位存储属性”的例子。在该例中，我们编写了一个枚举类型 FeatureSupports，用于列举编译器对各种特性的支持。而结构体 Compiler 则包含了一个 int 类型成员 spp。由于各种特性都具有“支持”和“不支持”两种状态，所以为了节省存储空间，我们让每个 FeatureSupports 的枚举值占据一个特定的比特位置，并在使用时通过“或”运算压缩地存储在 Compiler 的 spp 成员中（即 bitset 的概念）。在使用时，则可以通过检查 spp 的某位来判断编译器对特性是否支持。

有的时候这样的枚举值会非常多，而且还会在代码维护中不断增加。那么代码编写者必须想出办法来对这些枚举进行校验，比如查验一下是否有重位等。在本例中程序员的做法是使用一个“最大枚举”SMAX，并通过比较 SMAX - 1 与所有其他枚举的或运算值来验证是否有枚举值重位。可以想象，如果 SAssert 被误定义为 0x0001，表达式 (SMAX - 1) == (C99 | ExtInt | SAssert | NoExcept) 将不再成立。

在本例中我们使用了断言 assert。但 assert 是一个运行时的断言，这意味着不运行程序我们将无法得知是否有枚举重位。在一些情况下，这是不可接受的，因为可能单次运行代码并不会调用到 assert 相关的代码路径。因此这样的校验最好是在编译时期就能完成。

在一些 C++ 的模板的编写中，我们可能也会遇到相同的情况，比如下面这个例子，如代

码清单 2-8 所示。

代码清单 2-8

```
#include <cassert>
#include <cstring>
using namespace std;

template <typename T, typename U> int bit_copy(T& a, U& b){
    assert(sizeof(b) == sizeof(a));
    memcpy(&a,&b,sizeof(b));
};

int main() {
    int a = 0x2468;
    double b;
    bit_copy(a, b);
}
// 编译选项 :g++ 2-5-3.cpp
```

代码清单 2-8 中的 assert 是要保证 a 和 b 两种类型的长度一致，这样 bit_copy 才能够保证复制操作不会遇到越界等问题。这里我们还是使用 assert 的这样的运行时断言，但如果 bit_copy 不被调用，我们将无法触发该断言。实际上，正确产生断言的时机应该在模板实例化时，即编译时期。

代码清单 2-7 和代码清单 2-8 这类问题的解决方案是进行编译时期的断言，即所谓的“静态断言”。事实上，利用语言规则实现静态断言的讨论非常多，比较典型的实现是开源库 Boost 内置的 BOOST_STATIC_ASSERT 断言机制（利用 sizeof 操作符）。我们可以利用“除 0”会导致编译器报错这个特性来实现静态断言。

```
#define assert_static(e) \
    do { \
        enum { assert_static__ = 1/(e) }; \
    } while (0)
```

在理解这段代码时，读者可以忽略 do while 循环以及 enum 这些语法上的技巧。真正起作用的只是 1/(e) 这个表达式。把它应用到代码清单 2-8 中，就会得到代码清单 2-9。

代码清单 2-9

```
#include <cstring>
using namespace std;

#define assert_static(e) \
    do { \
        enum { assert_static__ = 1/(e) }; \
    } while (0)
```

```
template <typename T, typename U> int bit_copy(T& a, U& b){
    assert_static(sizeof(b) == sizeof(a));
    memcpy(&a,&b,sizeof(b));
};

int main() {
    int a = 0x2468;
    double b;
    bit_copy(a, b);
}
// 编译选项 :g++ -std=c++11 2-5-4.cpp
```

结果如我们预期的，在模板实例化时我们会得到编译器的错误报告，读者可以实验一下在自己本机运行的结果。在我们的实验机上会输出比较长的错误信息，主要信息是除零错误。当然，读者也可以尝试一下 Boost 库内置的 BOOST_STATIC_ASSERT，输出的主要信息是 sizeof 错误。但无论是哪种方式的静态断言，其缺陷都是很明显的：诊断信息不够充分，不熟悉该静态断言实现的程序员可能一时无法将错误对应到断言错误上，从而难以准确定位错误的根源。

在 C++11 标准中，引入了 static_assert 断言来解决这个问题。static_assert 使用起来非常简单，它接收两个参数，一个是断言表达式，这个表达式通常需要返回一个 bool 值；一个则是警告信息，它通常也就是一段字符串。我们可以用 static_assert 替换一下代码清单 2-9 中 bit_copy 的声明。

```
template <typename t, typename u> int bit_copy(t& a, u& b){
    static_assert(sizeof(b) == sizeof(a),"the parameters of bit_copy must have
    same width.");
};
```

那么再次编译代码清单 2-9 的时候，我们就会得到如下信息：

```
error: static assertion failed: "the parameters of bit_copy should have same width."
```

这样的错误信息就非常清楚，也非常有利于程序员排错。而由于 static_assert 是编译时期的断言，其使用范围不像 assert 一样受到限制。在通常情况下，static_assert 可以用于任何名字空间，如代码清单 2-10 所示。

代码清单 2-10

```
static_assert(sizeof(int) == 8, "This 64-bit machine should follow this!");
int main() { return 0; }
// 编译选项 :g++ -std=c++11 2-5-5.cpp
```

而在 C++ 中，函数则不可能像代码清单 2-10 中的 static_assert 这样独立于任何调用之外运行。因此将 static_assert 写在函数体外通常是较好的选择，这让代码阅读者可以较容易发

现 static_assert 为断言而非用户定义的函数。而反过来讲，必须注意的是，static_assert 的断言表达式的结果必须是在编译时期可以计算的表达式，即必须是常量表达式。如果读者使用了变量，则会导致错误，如代码清单 2-11 所示。

代码清单 2-11

```
int positive(const int n) {
    static_assert(n > 0, "value must >0");
}
// 编译选项 :g++ -std=c++11 -c 2-5-6.cpp
```

代码清单 2-11 使用了参数变量 n（虽然是个 const 参数），因而 static_assert 无法通过编译。对于此例，如果程序员需要的只是运行时的检查，那么还是应该使用 assert 宏。

2.6 noexcept 修饰符与 noexcept 操作符

☞ 类别：库作者

相比于断言适用于排除逻辑上不可能存在的状态，异常通常是用于逻辑上可能发生的错误。在 C++98 中，我们看到了一套完整的不同于 C 的异常处理系统。通过这套异常处理系统，C++ 拥有了远比 C 强大的异常处理功能。

在异常处理的代码中，程序员有可能看到过如下的异常声明表达形式：

```
void excpt_func() throw(int, double) { ... }
```

在 excpt_func 函数声明之后，我们定义了一个动态异常声明 throw(int, double)，该声明指出了 excpt_func 可能抛出的异常的类型。事实上，该特性很少被使用，因此在 C++11 中被弃用了（参见附录 B），而表示函数不会抛出异常的动态异常声明 throw() 也被新的 noexcept 异常声明所取代。

noexcept 形如其名地，表示其修饰的函数不会抛出异常。不过与 throw() 动态异常声明不同的是，在 C++11 中如果 noexcept 修饰的函数抛出了异常，编译器可以选择直接调用 std::terminate() 函数来终止程序的运行，这比基于异常机制的 throw() 在效率上会高一些。这是因为异常机制会带来一些额外开销，比如函数抛出异常，会导致函数栈被依次地展开（unwind），并依帧调用在本帧中已构造的自动变量的析构函数等。

从语法上讲，noexcept 修饰符有两种形式，一种就是简单地在函数声明后加上 noexcept 关键字。比如：

```
void excpt_func() noexcept;
```

另外一种则可以接受一个常量表达式作为参数，如下所示：

```
void excpt_func() noexcept （常量表达式）;
```

常量表达式的结果会被转换成一个bool类型的值。该值为true，表示函数不会抛出异常，反之，则有可能抛出异常。这里，不带常量表达式的noexcept相当于声明了noexcept(true)，即不会抛出异常。

在通常情况下，在C++11中使用noexcept可以有效地阻止异常的传播与扩散。我们可以看看下面这个例子，如代码清单2-12所示。

代码清单2-12

```
#include <iostream>
using namespace std;
void Throw() { throw 1; }
void NoBlockThrow() { Throw(); }
void BlockThrow() noexcept { Throw(); }

int main() {
    try {
        Throw();
    }
    catch(...) {
        cout << "Found throw." << endl;      // Found throw.
    }

    try {
        NoBlockThrow();
    }
    catch(...) {
        cout << "Throw is not blocked." << endl;     // Throw is not blocked.
    }

    try {
        BlockThrow();    // terminate called after throwing an instance of 'int'
    }
    catch(...) {
        cout << "Found throw 1." << endl;
    }
}
// 编译选项 :g++ -std=c++11 2-6-1.cpp
```

在代码清单2-12中，我们定义了Throw函数，该函数的唯一作用是抛出一个异常。而NoBlockThrow是一个调用Throw的普通函数，BlockThrow则是一个noexcept修饰的函数。从main的运行中我们可以看到，NoBlockThrow会让Throw函数抛出的异常继续抛出，直到main中的catch语句将其捕捉。而BlockThrow则会直接调用std::terminate中断程序的执行，从而阻止了异常的继续传播。从使用效果上看，这与C++98中的throw()是一样的。

而noexcept作为一个操作符时，通常可以用于模板。比如：

```
template <class T>
  void fun() noexcept(noexcept(T())) {}
```

这里，fun 函数是否是一个 noexcept 的函数，将由 T() 表达式是否会抛出异常所决定。这里的第二个 noexcept 就是一个 noexcept 操作符。当其参数是一个有可能抛出异常的表达式的时候，其返回值为 false，反之为 true（实际 noexcept 参数返回 false 还包括一些情况，这里就不展开讲了）。这样一来，我们就可以使模板函数根据条件实现 noexcept 修饰的版本或无 noexcept 修饰的版本。从泛型编程的角度看来，这样的设计保证了关于“函数是否抛出异常”这样的问题可以通过表达式进行推导。因此这也可以视作 C++11 为了更好地支持泛型编程而引入的特性。

虽然 noexcept 修饰的函数通过 std::terminate 的调用来结束程序的执行的方式可能会带来很多问题，比如无法保证对象的析构函数的正常调用，无法保证栈的自动释放等，但很多时候，“暴力”地终止整个程序确实是很简单有效的做法。事实上，noexcept 被广泛地、系统地应用在 C++11 的标准库中，用于提高标准库的性能，以及满足一些阻止异常扩散的需求。

比如在 C++98 中，存在着使用 throw() 来声明不抛出异常的函数。

```
template<class T> class A {
  public:
    static constexpr T min() throw() { return T(); }
    static constexpr T max() throw() { return T(); }
    static constexpr T lowest() throw() { return T(); }
...
```

而在 C++11 中，则使用 noexcept 来替换 throw()。

```
template<class T> class A {
  public:
    static constexpr T min() noexcept { return T(); }
    static constexpr T max() noexcept  { return T(); }
    static constexpr T lowest() noexcept { return T(); }
...
```

又比如，在 C++98 中，new 可能会包含一些抛出的 std::bad_alloc 异常。

```
void* operator new(std::size_t) throw(std::bad_alloc);
void* operator new[](std::size_t) throw(std::bad_alloc);
```

而在 C++11 中，则使用 noexcept(false) 来进行替代。

```
void* operator new(std::size_t) noexcept(false);
void* operator new[](std::size_t) noexcept(false);
```

当然，noexcept 更大的作用是保证应用程序的安全。比如一个类析构函数不应该抛出异常，那么对于常被析构函数调用的 delete 函数来说，C++11 默认将 delete 函数设置成 noexcept，就可以提高应用程序的安全性。

```
void operator delete(void*) noexcept;
void operator delete[](void*) noexcept;
```

而同样出于安全考虑，C++11 标准中让类的析构函数默认也是 noexcept(true) 的。当然，如果程序员显式地为析构函数指定了 noexcept，或者类的基类或成员有 noexcept(false) 的析构函数，析构函数就不会再保持默认值。我们可以看看下面的例子，如代码清单 2-13 所示。

代码清单 2-13

```
#include <iostream>
using namespace std;

struct A {
    ~A() { throw 1; }
};

struct B {
    ~B() noexcept(false) { throw 2; }
};

struct C {
    B b;
};

int funA() { A a; }
int funB() { B b; }
int funC() { C c; }

int main() {
    try {
        funB();
    }
    catch(...){
        cout << "caught funB." << endl; // caught funB.
    }

    try {
        funC();
    }
    catch(...){
        cout << "caught funC." << endl; // caught funC.
    }

    try {
        funA(); // terminate called after throwing an instance of 'int'
    }
    catch(...){
        cout << "caught funA." << endl;
    }
}
// 编译选项 :g++ -std=c++11 2-6-2.cpp
```

在代码清单 2-13 中，无论是析构函数声明为 noexcept(false) 的类 B，还是包含了 B 类型成员的类 C，其析构函数都是可以抛出异常的。只有什么都没有声明的类 A，其析构函数被默认为 noexcept(true)，从而阻止了异常的扩散。这在实际的使用中，应该引起程序员的注意。

2.7 快速初始化成员变量

☞ **类别：部分人**

在 C++98 中，支持了在类声明中使用等号“=”加初始值的方式，来初始化类中静态成员常量。这种声明方式我们也称之为“就地”声明。就地声明在代码编写时非常便利，不过 C++98 对类中就地声明的要求却非常高。如果静态成员不满足常量性，则不可以就地声明，而且即使常量的静态成员也只能是整型或者枚举型才能就地初始化。而非静态成员变量的初始化则必须在构造函数中进行。我们来看看下面的例子，如代码清单 2-14 所示。

代码清单 2-14

```
class Init{
public:
    Init(): a(0){}
    Init(int d): a(d){}
private:
    int a;
    const static int b = 0;
    int c = 1;           // 成员，无法通过编译
    static int d = 0;    // 成员，无法通过编译
    static const double e = 1.3;         // 非整型或者枚举，无法通过编译
    static const char * const f = "e";   // 非整型或者枚举，无法通过编译
};
// 编译选项 :g++ -c 2-7-1.cpp
```

在代码清单 2-14 中，成员 c、静态成员 d、静态常量成员 e 以及静态常量指针 f 的就地初始化都无法通过编译（这里，使用 g++ 的读者可能发现就地初始化 double 类型静态常量 e 是可以通过编译的，不过这实际是 GNU 对 C++ 的一个扩展，并不遵从 C++ 标准）。在 C++11 中，标准允许非静态成员变量的初始化有多种形式。具体而言，除了初始化列表外，在 C++11 中，标准还允许使用等号 = 或者花括号 {} 进行就地的非静态成员变量初始化。比如：

```
struct init{ int a = 1; double b {1.2}; };
```

在这个名叫 init 的结构体中，我们给了非静态成员 a 和 b 分别赋予初值 1 和 1.2。这在 C++11 中是一个合法的结构体声明。虽然这里采用的一对花括号 {} 的初始化方法读者第一次见到，不过在第 3 章中，读者会在 C++ 对于初始化表达式的改动发现，花括号式的集合

（列表）初始化已经成为 C++11 中初始化声明的一种通用形式，而其效果类似于 C++98 中使用圆括号 () 对自定义变量的表达式列表初始化。不过在 C++11 中，对于非静态成员进行就地初始化，两者却并非等价的，如代码清单 2-15 所示。

代码清单 2-15

```
#include <string>
using namespace std;

struct C {
    C(int i):c(i){};
    int c;
};

struct init {
    int a = 1;
    string b("hello");  // 无法通过编译
    C c(1);             // 无法通过编译
};
// 编译选项 :g++ -std=c++11 -c 2-7-2.cpp
```

从代码清单 2-15 中可以看到，就地圆括号式的表达式列表初始化非静态成员 b 和 c 都会导致编译出错。

在 C++11 标准支持了就地初始化非静态成员的同时，初始化列表这个手段也被保留下来了。如果两者都使用，是否会发生冲突呢？我们来看下面这个例子，如代码清单 2-16 所示。

代码清单 2-16

```
#include <iostream>
using namespace std;

struct Mem {
    Mem() { cout << "Mem default, num: " << num << endl; }
    Mem(int i): num(i) { cout << "Mem, num: " << num << endl; }

    int num = 2;  // 使用 = 初始化非静态成员
};

class Group {
public:
    Group() { cout << "Group default. val: " << val << endl; }
    Group(int i): val('G'), a(i) { cout << "Group. val: " << val << endl; }
    void NumOfA() { cout << "number of A: " << a.num << endl; }
    void NumOfB() { cout << "number of B: " << b.num << endl; }

private:
    char val{'g'};  // 使用 {} 初始化非静态成员
```

```
    Mem a;
    Mem b{19};        // 使用{}初始化非静态成员
};

int main() {
    Mem member;           // Mem default, num: 2

    Group group;          // Mem default, num: 2
                          // Mem, num: 19
                          // Group default. val: g

    group.NumOfA();       // number of A: 2
    group.NumOfB();       // number of B: 19

    Group group2(7);      // Mem, num: 7
                          // Mem, num: 19
                          // Group. val: G

    group2.NumOfA();      // number of A: 7
    group2.NumOfB();      // number of B: 19
}
// 编译选项 :g++ 2-7-3.cpp -std=c++11
```

在代码清单2-16中，我们定义了有两个初始化函数的类Mem，此外还定义了包含两个Mem对象的Group类。类Mem中的成员变量num，以及class Group中的成员变量a、b、val，采用了与C++98完全不同的初始化方式。读者可以从main函数的打印输出中看到，就地初始化和初始化列表并不冲突。程序员可以为同一成员变量既声明就地的列表初始化，又在初始化列表中进行初始化，只不过初始化列表总是看起来“后作用于”非静态成员。也就是说，初始化列表的效果总是优先于就地初始化的。

相对于传统的初始化列表，在类声明中对非静态成员变量进行就地列表初始化可以降低程序员的工作量。当然，我们只在有多个构造函数，且有多个成员变量的时候可以看到新方式带来的便利。我们来看看下面的例子，如代码清单2-17所示。

代码清单 2-17

```
#include <string>
using namespace std;

class Mem {
public:
    Mem(int i): m(i){}

private:
    int m;
};
```

```
class Group {
public:
    Group(){}                          // 这里就不需要初始化 data、mem、name 成员了
    Group(int a): data(a) {}           // 这里就不需要初始化 mem、name 成员了
    Group(Mem m) : mem(m) {}           // 这里就不需要初始化 data、name 成员了
    Group(int a, Mem m, string n): data(a), mem(m), name(n){}

private:
    int data = 1;
    Mem mem{0};
    string name{"Group"};
};
// 编译选项 :g++ 2-7-4.cpp -std=c++11 -c
```

在代码清单 2-17 中，Group 有 4 个构造函数。可以想象，如果我们使用的是 C++98 的编译器，我们不得不在 Group()、Group(int a)，以及 Group(Mem m) 这 3 个构造函数中将 data、mem、name 这 3 个成员都写进初始化列表。但如果使用的是 C++11 的编译器，那么通过对非静态成员变量的就地初始化，我们就可以避免重复地在初始化列表中写上每个非静态成员了（在 C++98 中，我们还可以通过调用公共的初始化函数来达到类似的目的，不过目前在书写的复杂性及效率性上远低于 C++11 改进后的做法）。

此外，值得注意的是，对于非常量的静态成员变量，C++11 则与 C++98 保持了一致。程序员还是需要到头文件以外去定义它，这会保证编译时，类静态成员的定义最后只存在于一个目标文件中。不过对于静态常量成员，除了 const 关键字外，在本书第 6 章中我们会看到还可以使用 constexpr 来对静态常量成员进行声明。

2.8 非静态成员的 sizeof

☞ **类别：部分人**

从 C 语言被发明开始，sizeof 就是一个运算符，也是 C 语言中除了加减乘除以外为数不多的特殊运算符之一。而在 C++ 引入类（class）类型之后，sizeof 的定义也随之进行了拓展。不过在 C++98 标准中，对非静态成员变量使用 sizeof 是不能够通过编译的。我们可以看看下面的例子，如代码清单 2-18 所示。

代码清单 2-18

```
#include <iostream>
using namespace std;

struct People {
public:
    int hand;
    static People * all;
```

```
};

int main() {
    People p;
    cout << sizeof(p.hand) << endl;          // C++98 中通过，C++11 中通过
    cout << sizeof(People::all) << endl;     // C++98 中通过，C++11 中通过
    cout << sizeof(People::hand) << endl;    // C++98 中错误，C++11 中通过
}
// 编译选项 :g++ 2-8-1.cpp
```

注意最后一个 sizeof 操作。在 C++11 中，对非静态成员变量使用 sizeof 操作是合法的。而在 C++98 中，只有静态成员，或者对象的实例才能对其成员进行 sizeof 操作。因此如果读者只有一个支持 C++98 标准的编译器，在没有定义类实例的时候，要获得类成员的大小，我们通常会采用以下的代码：

```
sizeof(((People*)0)->hand);
```

这里我们强制转换 0 为一个 People 类的指针，继而通过指针的解引用获得其成员变量，并用 sizeof 求得该成员变量的大小。而在 C++11 中，我们无需这样的技巧，因为 sizeof 可以作用的表达式包括了类成员表达式。

```
sizeof(People::hand);
```

可以看到，无论从代码的可读性还是编写的便利性，C++11 的规则都比强制指针转换的方案更胜一筹。

2.9 扩展的 friend 语法

☞ **类别：部分人**

friend 关键字在 C++ 中是一个比较特别的存在。因为我们常常会发现，一些面向对象程序语言，比如 Java，就没有定义 friend 关键字。friend 关键字用于声明类的友元，友元可以无视类中成员的属性。无论成员是 public、protected 或是 private 的，友元类或友元函数都可以访问，这就完全破坏了面向对象编程中封装性的概念。因此，使用 friend 关键字充满了争议性。在通常情况下，面向对象程序开发的专家会建议程序员使用 Get/Set 接口来访问类的成员，但有的时候，friend 关键字确实会让程序员少写很多代码。因此即使存在争论，friend 还是在很多程序中被使用到。而 C++11 对 friend 关键字进行了一些改进，以保证其更加好用。我们可以看看下面的例子，如代码清单 2-19 所示。

代码清单 2-19

```
class Poly;
typedef Poly P;

class LiLei {
```

```
    friend class Poly;   // C++98 通过，C++11 通过
};

class Jim {
    friend Poly;         // C++98 失败，C++11 通过
};

class HanMeiMei {
    friend P;            // C++98 失败，C++11 通过
};
// 编译选项 :g++ -std=c++11 2-9-1.cpp
```

在代码清单 2-19 中，我们声明了 3 个类型：LiLei、Jim 和 HanMeiMei，它们都有一个友元类型 Poly。从编译通过与否的状况中我们可以看出，在 C++11 中，声明一个类为另外一个类的友元时，不再需要使用 class 关键字。本例中的 Jim 和 HanMeiMei 就是这样一种情况，在 HanMeiMei 的声明中，我们甚至还使用了 Poly 的别名 P，这同样是可行的。

虽然在 C++11 中这是一个小的改进，却会带来一点应用的变化——程序员可以为类模板声明友元了。这在 C++98 中是无法做到的。比如下面这个例子，如代码清单 2-20 所示。

代码清单 2-20

```
class P;
template <typename T> class People {
    friend T;
};

People<P> PP;    // 类型 P 在这里是 People 类型的友元
People<int> Pi; // 对于 int 类型模板参数，友元声明被忽略
// 编译选项 :g++ -std=c++11 2-9-2.cpp
```

从代码清单 2-20 中我们看到，对于 People 这个模板类，在使用类 P 为模板参数时，P 是 People<P> 的一个 friend 类。而在使用内置类型 int 作为模板参数的时候，People<int> 会被实例化为一个普通的没有友元定义的类型。这样一来，我们就可以在模板实例化时才确定一个模板类是否有友元，以及谁是这个模板类的友元。这是一个非常有趣的小特性，在编写一些测试用例的时候，使用该特性是很有好处的。我们看看下面的例子，该例子源自一个实际的测试用例，如代码清单 2-21 所示。

代码清单 2-21

```
// 为了方便测试，进行了危险的定义
#ifdef UNIT_TEST
#define private public
#endif
class Defender {
```

```
public:
    void Defence(int x, int y){}
    void Tackle(int x, int y){}

private:
    int pos_x = 15;
    int pos_y = 0;
    int speed = 2;
    int stamina = 120;
};

class Attacker {
public:
    void Move(int x, int y){}
    void SpeedUp(float ratio){}

private:
    int pos_x = 0;
    int pos_y = -30;
    int speed = 3;
    int stamina = 100;
};

#ifdef UNIT_TEST
class Validator {
public:
    void Validate(int x, int y, Defender & d){}
    void Validate(int x, int y, Attacker & a){}
};

int main() {
    Defender d;
    Attacker a;
    a.Move(15, 30);
    d.Defence(15, 30);
    a.SpeedUp(1.5f);
    d.Defence(15, 30);
    Validator v;
    v.Validate(7, 0, d);
    v.Validate(1, -10, a);
    return 0;
}
#endif
// 编译选项 :g++ 2-9-3.cpp -std=c++11 -DUNIT_TEST
```

在代码清单 2-21 所示的这个例子中，测试人员的目的是在一系列函数调用后，检查 Attacker 类变量 a 和 Defender 类变量 d 中成员变量的值是否符合预期。这里，按照封装的思想，所有成员变量被声明为 private 的。但 Attacker 和 Defender 的开发者为了方便，并没有为每个成员写 Get 函数，也没有为 Attacker 和 Defender 增加友元定义。而测试人员为了能够

快速写出测试程序，采用了比较危险的做法，即使用宏将 private 关键字统一替换为 public 关键字。这样一来，类中的 private 成员就都成了 public 的。这样的做法存在 4 个缺点：一是如果侥幸程序中没有变量包含 private 字符串，该方法可以正常工作，但反之，则有可能导致严重的编译时错误；二是这种做法会降低代码可读性，因为改变了一个常见关键字的意义，没有注意到这个宏的程序员可能会非常迷惑程序的行为；三是如果在类的成员定义时不指定关键字（如 public、private、protect 等），而使用默认的 private 成员访问限制，那么该方法也不能达到目的；四则很简单，这样的做法看起来也并不漂亮。

不过由于有了扩展的 friend 声明，在 C++11 中，我们可以将 Defender 和 Attacker 类改良一下。我们看看下面的例子，如代码清单 2-22 所示。

代码清单 2-22

```
template <typename T> class DefenderT {
public:
    friend T;
    void Defence(int x, int y){}
    void Tackle(int x, int y){}

private:
    int pos_x = 15;
    int pos_y = 0;
    int speed = 2;
    int stamina = 120;
};

template <typename T> class AttackerT {
public:
    friend T;
    void Move(int x, int y){}
    void SpeedUp(float ratio){}

private:
    int pos_x = 0;
    int pos_y = -30;
    int speed = 3;
    int stamina = 100;
};

using Defender = DefenderT<int>;     // 普通的类定义，使用 int 做参数
using Attacker = AttackerT<int>;

#ifdef UNIT_TEST
class Validator {
public:
    void Validate(int x, int y, DefenderTest & d){}
    void Validate(int x, int y, AttackerTest & a){}
```

```
};

using DefenderTest = DefenderT<Validator>;   // 测试专用的定义，Validator 类成为友元
using AttackerTest = AttackerT<Validator>;

int main() {
    DefenderTest d;
    AttackerTest a;
    a.Move(15, 30);
    d.Defence(15, 30);
    a.SpeedUp(1.5f);
    d.Defence(15, 30);
    Validator v;
    v.Validate(7, 0, d);
    v.Validate(1, -10, a);
    return 0;
}
#endif
// 编译选项 :g++ 2-9-4.cpp -std=c++11 -DUNIT_TEST
```

在代码清单 2-22 中，我们把原有的 Defender 和 Attacker 类定义为模板类 DefenderT 和 AttackerT。而在需要进行测试的时候，我们使用 Validator 为模板参数，实例化出 DefenderTest 及 AttackerTest 版本的类，这个版本的特点是，Validator 是它们的友元，可以任意访问任何成员函数。而另外一个版本则是使用 int 类型进行实例化的 Defender 和 Attacker，按照 C++11 的定义，它们不会有友元。因此这个版本保持了良好的封装性，可以用于提供接口用于常规使用。

值得注意的是，在代码清单 2-22 中，我们使用了 using 来定义类型的别名，这跟使用 typedef 的定义类型的别名是完全一样的。使用 using 定义类型别名是 C++11 中的一个新特性，我们可以在 3.10 节中看到相关的描述。

2.10 final/override 控制

☞ **类别：部分人**

在了解 C++11 中的 final/override 关键字之前，我们先回顾一下 C++ 关于重载的概念。简单地说，一个类 A 中声明的虚函数 fun 在其派生类 B 中再次被定义，且 B 中的函数 fun 跟 A 中 fun 的原型一样（函数名、参数列表等一样），那么我们就称 B 重载（overload）了 A 的 fun 函数。对于任何 B 类型的变量，调用成员函数 fun 都是调用了 B 重载的版本。而如果同时有 A 的派生类 C，却并没有重载 A 的 fun 函数，那么调用成员函数 fun 则会调用 A 中的版本。这在 C++ 中就实现多态。

在通常情况下，一旦在基类 A 中的成员函数 fun 被声明为 virtual 的，那么对于其派生类

B 而言，fun 总是能够被重载的（除非被重写了）。有的时候我们并不想 fun 在 B 类型派生类中被重载，那么，C++98 没有方法对此进行限制。我们看看下面这个具体的例子，如代码清单 2-23 所示。

代码清单 2-23

```
#include <iostream>
using namespace std;

class MathObject{
public:
    virtual double Arith() = 0;
    virtual void Print() = 0;
};

class Printable : public MathObject{
public:
   double Arith() = 0;
   void Print() // 在 C++98 中我们无法阻止该接口被重写
   {
       cout << "Output is: " << Arith() << endl;
   }
};

class Add2 : public Printable {
public:
    Add2(double a, double b): x(a), y(b) {}
    double Arith() { return x + y; }
private:
    double x, y;
};

class Mul3 : public Printable {
public:
    Mul3(double a, double b, double c): x(a), y(b), z(c) {}
    double Arith() { return x * y * z; }
private:
    double x, y, z;
};
// 编译选项 :g++ 2-10-1.cpp
```

在代码清单 2-23 中，我们的基础类 MathObject 定义了两个接口：Arith 和 Print。类 Printable 则继承于 MathObject 并实现了 Print 接口。接下来，Add2 和 Mul3 为了使用 MathObject 的接口和 Printable 的 Print 的实现，于是都继承了 Printable。这样的类派生结构，在面向对象的编程中非常典型。不过倘若这里的 Printable 和 Add2 是由两个程序员完成的，Printable 的编写者不禁会有一些忧虑，如果 Add2 的编写者重载了 Print 函数，那么他所期望的统一风格的打印方式将不复存在。

对于 Java 这种所有类型派生于单一元类型（Object）的语言来说，这种问题早就出现了。因此 Java 语言使用了 final 关键字来阻止函数继续重写。final 关键字的作用是使派生类不可覆盖它所修饰的虚函数。C++11 也采用了类似的做法，如代码清单 2-24 所示的例子。

代码清单 2-24

```
struct Object{
    virtual void fun() = 0;
};

struct Base : public Object {
    void fun() final;    // 声明为 final
};

struct Derived : public Base {
    void fun();      // 无法通过编译
};
// 编译选项 :g++ -c -std=c++11 2-10-2.cpp
```

在代码清单 2-24 中，派生于 Object 的 Base 类重载了 Object 的 fun 接口，并将本类中的 fun 函数声明为 final 的。那么派生于 Base 的 Derived 类对接口 fun 的重载则会导致编译时的错误。同理，在代码清单 2-23 中，Printable 的编写者如果要阻止派生类重载 Print 函数，只需要在定义时使用 final 进行修饰就可以了。

读者可能注意到了，在代码清单 2-23 及代码清单 2-24 两个例子当中，final 关键字都是用于描述一个派生类的。那么基类中的虚函数是否可以使用 final 关键字呢？答案是肯定的，不过这样将使用该虚函数无法被重载，也就失去了虚函数的意义。如果不想成员函数被重载，程序员可以直接将该成员函数定义为非虚的。而 final 通常只在继承关系的“中途”终止派生类的重载中有意义。从接口派生的角度而言，final 可以在派生过程中任意地阻止一个接口的可重载性，这就给面向对象的程序员带来了更大的控制力。

在 C++ 中重载还有一个特点，就是对于基类声明为 virtual 的函数，之后的重载版本都不需要再声明该重载函数为 virtual。即使在派生类中声明了 virtual，该关键字也是编译器可以忽略的。这带来了一些书写上的便利，却带来了一些阅读上的困难。比如代码清单 2-23 中的 Printable 的 Print 函数，程序员无法从 Printable 的定义中看出 Print 是一个虚函数还是非虚函数。另外一点就是，在 C++ 中有的虚函数会“跨层”，没有在父类中声明的接口有可能是祖先的虚函数接口。比如在代码清单 2-23 中，如果 Printable 不声明 Arith 函数，其接口在 Add2 和 Mul3 中依然是可重载的，这同样是在父类中无法读到的信息。这样一来，如果类的继承结构比较长（不断地派生）或者比较复杂（比如偶尔多重继承），派生类的编写者会遇到信息分散、难以阅读的问题（虽然有时候编辑器会进行提示，不过编辑器不是总是那么有效）。而自己是否在重载一个接口，以及自己重载的接口的名字是否有拼写错误等，都非常

不容易检查。

在 C++11 中为了帮助程序员写继承结构复杂的类型，引入了虚函数描述符 override，如果派生类在虚函数声明时使用了 override 描述符，那么该函数必须重载其基类中的同名函数，否则代码将无法通过编译。我们来看一下如代码清单 2-25 所示的这个简单的例子。

代码清单 2-25

```
struct Base {
    virtual void Turing() = 0;
    virtual void Dijkstra() = 0;
    virtual void VNeumann(int g) = 0;
    virtual void DKnuth() const;
    void Print();
};

struct DerivedMid: public Base {
    // void VNeumann(double g);
    // 接口被隔离了，曾想多一个版本的 VNeumann 函数
};

struct DerivedTop : public DerivedMid {
    void Turing() override;
    void Dikjstra() override;              // 无法通过编译，拼写错误，并非重载
    void VNeumann(double g) override;      // 无法通过编译，参数不一致，并非重载
    void DKnuth() override;                // 无法通过编译，常量性不一致，并非重载
    void Print() override;                 // 无法通过编译，非虚函数重载
};
// 编译选项 :g++ -c -std=c++11 2-10-3.cpp
```

在代码清单 2-25 中，我们在基类 Base 中定义了一些 virtual 的函数（接口）以及一个非 virtual 的函数 Print。其派生类 DerivedMid 中，基类的 Base 的接口都没有重载，不过通过注释可以发现，DerivedMid 的作者曾经想要重载出一个“void VNeumann(double g)”的版本。这行注释显然迷惑了编写 DerivedTop 的程序员，所以 DerivedTop 的作者在重载所有 Base 类的接口的时候，犯下了 3 种不同的错误：

- 函数名拼写错，Dijkstra 误写作了 Dikjstra。
- 函数原型不匹配，VNeumann 函数的参数类型误做了 double 类型，而 DKnuth 的常量性在派生类中被取消了。
- 重写了非虚函数 Print。

如果没有 override 修饰符，DerivedTop 的作者可能在编译后都没有意识到自己犯了这么多错误。因为编译器对以上 3 种错误不会有任何的警示。这里 override 修饰符则可以保证编译器辅助地做一些检查。我们可以看到，在代码清单 2-25 中，DerivedTop 作者的 4 处错误都无法通过编译。

此外，值得指出的是，在 C++ 中，如果一个派生类的编写者自认为新写了一个接口，而实际上却重载了一个底层的接口（一些简单的名字如 get、set、print 就容易出现这样的状况），出现这种情况编译器还是爱莫能助的。不过这样无意中的重载一般不会带来太大的问题，因为派生类的变量如果调用了该接口，除了可能存在的一些虚函数开销外，仍然会执行派生类的版本。因此编译器也就没有必要提供检查“非重载”的状况。而检查“一定重载”的 override 关键字，对程序员的实际应用则会更有意义。

还有值得注意的是，如我们在第 1 章中提到的，final/override 也可以定义为正常变量名，只有在其出现在函数后时才是能够控制继承 / 派生的关键字。通过这样的设计，很多含有 final/override 变量或者函数名的 C++98 代码就能够被 C++ 编译器编译通过了。但出于安全考虑，建议读者在 C++11 代码中应该尽可能地避免这样的变量名称或将其定义在宏中，以防发生不必要的错误。

2.11 模板函数的默认模板参数

☞ **类别：所有人**

在 C++11 中模板和函数一样，可以有默认的参数。这就带来了一定的复杂性。我可以通过代码清单 2-26 所示的这个简单的模板函数的例子来回顾一下函数模板的定义。

代码清单 2-26

```
#include <iostream>
using namespace std;

// 定义一个函数模板
template <typename T> void TempFun(T a) {
    cout << a << endl;
}

int main() {
    TempFun(1);      // 1, (实例化为 TempFun<const int>(1))
    TempFun("1");    // 1, (实例化为 TempFun<const char *>("1"))
}
// 编译选项 :g++ 2-11-1.cpp
```

在代码清单 2-26 中，当编译器解析到函数调用 fun(1) 的时候，发现 fun 是一个函数模板。这时候编译器就会根据实参 1 的类型 const int 推导实例化出模板函数 void TempFun<const int>(int)，再进行调用。相应的，对于 fun("1") 来说也是类似的，不过编译器实例化出的模板函数的参数的类型将是 const char *。

函数模板在 C++98 中与类模板一起被引入，不过在模板类声明的时候，标准允许其有默认模板参数。默认的模板参数的作用好比函数的默认形参。然而由于种种原因，C++98 标准

却不支持函数模板的默认模板参数。不过在 C++11 中，这一限制已经被解除了，我们可以看看下面这个例子，如代码清单 2-27 所示。

代码清单 2-27

```
void DefParm(int m = 3) {}  // c++98 编译通过，c++11 编译通过
template <typename T = int>
    class DefClass {};      // c++98 编译通过，c++11 编译通过
template <typename T = int>
    void DefTempParm() {};  // c++98 编译失败，c++11 编译通过
// 编译选项 :g++ -c -std=c++11 2-11-1.cpp
```

可以看到，DefTempParm 函数模板拥有一个默认参数。使用仅支持 C++98 的编译器编译，DefTempParm 的编译会失败，而支持 C++11 的编译器则毫无问题。不过在语法上，与类模板有些不同的是，在为多个默认模板参数声明指定默认值的时候，程序员必须遵照“从右往左”的规则进行指定。而这个条件对函数模板来说并不是必须的，如代码清单 2-28 所示。

代码清单 2-28

```
template<typename T1, typename T2 = int> class DefClass1;
template<typename T1 = int, typename T2> class DefClass2;   // 无法通过编译

template<typename T, int i = 0> class DefClass3;
template<int i = 0, typename T> class DefClass4;            // 无法通过编译

template<typename T1 = int, typename T2> void DefFunc1(T1 a, T2 b);
template<int i = 0, typename T> void DefFunc2(T a);
// 编译选项 :g++ -c -std=c++11 2-11-2.cpp
```

从代码清单 2-28 中可以看到，不按照从右往左定义默认类模板参数的模板类 DefClass2 和 DefClass4 都无法通过编译。而对于函数模板来说，默认模板参数的位置则比较随意。可以看到 DefFunc1 和 DefFunc2 都为第一个模板参数定义了默认参数，而第二个模板参数的默认值并没有定义，C++11 编译器却认为没有问题。

函数模板的参数推导规则也并不复杂。简单地讲，如果能够从函数实参中推导出类型的话，那么默认模板参数就不会被使用，反之，默认模板参数则可能会被使用。我们可以看看下面这个来自于 C++11 标准草案的例子，如代码清单 2-29 所示。

代码清单 2-29

```
template <class T, class U = double>
void f(T t = 0, U u = 0);

void g() {
```

```
    f(1, 'c');        // f<int,char>(1,'c')
    f(1);             // f<int,double>(1,0), 使用了默认模板参数 double
    f();              // 错误 : T 无法被推导出来
    f<int>();         // f<int,double>(0,0), 使用了默认模板参数 double
    f<int,char>();    // f<int,char>(0,0)
}
// 编译选项 :g++ -std=c++11 2-11-3.cpp
```

在代码清单 2-29 中，我们定义了一个函数模板 f，f 同时使用了默认模板参数和默认函数参数。可以看到，由于函数的模板参数可以由函数的实参推导而出，所以在 f(1) 这个函数调用中，我们实例化出了模板函数的调用应该为 f<int,double>(1,0)，其中，第二个类型参数 U 使用了默认的模板类型参数 double，而函数实参则为默认值 0。类似地，f<int>() 实例化出的模板函数第二参数类型为 double，值为 0。而表达式 f() 由于第一类型参数 T 的无法推导，从而导致了编译的失败。而通过这个例子我们也可以看到，默认模板参数通常是需要跟默认函数参数一起使用的。

还有一点应该强调一下，模板函数的默认形参不是模板参数推导的依据。函数模板参数的选择，总是由函数的实参推导而来的，这点读者在使用中应当注意。

2.12 外部模板

☞ **类别：部分人**

2.12.1 为什么需要外部模板

“外部模板”是 C++11 中一个关于模板性能上的改进。实际上，“外部”（extern）这个概念早在 C 的时候已经就有了。通常情况下，我们在一个文件中 a.c 中定义了一个变量 int i，而在另外一个文件 b.c 中想使用它，这个时候我们就会在没有定义变量 i 的 b.c 文件中做一个外部变量的声明。比如：

```
extern int i;
```

这样做的好处是，在分别编译了 a.c 和 b.c 之后，其生成的目标文件 a.o 和 b.o 中只有 i 这个符号[⊖]的一份定义。具体地，a.o 中的 i 是实在存在于 a.o 目标文件的数据区中的数据，而在 b.o 中，只是记录了 i 符号会引用其他目标文件中数据区中的名为 i 的数据。这样一来，在链接器（通常由编译器代为调用）将 a.o 和 b.o 链接成单个可执行文件（或者库文件）c 的时候，c 文件的数据区也只会有一个 i 的数据（供 a.c 和 b.c 的代码共享）。

而如果 b.c 中我们声明 int i 的时候不加上 extern 的话，那么 i 就会实实在在地既存在于 a.o 的数据区中，也存在于 b.o 的数据区中。那么链接器在链接 a.o 和 b.o 的时候，就会报告

⊖ 符号（symbol）是编译器 / 链接器的术语，读者可以简单地将它想象为一个变量名字。

错误，因为无法决定相同的符号是否需要合并。

而对于函数模板来说，现在我们遇到的几乎是一模一样的问题。不同的是，发生问题的不是变量（数据），而是函数（代码）。这样的困境是由于模板的实例化带来的。

注意 这里我们以函数模板为例，因为其只涉及代码，讲解起来比较直观。如果是类模板，则有可能涉及数据，不过其原理都是类似的。

比如，我们在一个 test.h 的文件中声明了如下一个模板函数：

```
template <typename T> void fun(T) {}
```

在第一个 test1.cpp 文件中，我们定义了以下代码：

```
#include "test.h"
void test1() { fun(3); }
```

而在另一个 test2.cpp 文件中，我们定义了以下代码：

```
#include "test.h"
void test2() { fun(4); }
```

由于两个源代码使用的模板函数的参数类型一致，所以在编译 test1.cpp 的时候，编译器实例化出了函数 fun<int>(int)，而当编译 test2.cpp 的时候，编译器又再一次实例化出了函数 fun<int>(int)。那么可以想象，在 test1.o 目标文件和 test2.o 目标文件中，会有两份一模一样的函数 fun<int>(int) 代码。

代码重复和数据重复不同。数据重复，编译器往往无法分辨是否是要共享的数据；而代码重复，为了节省空间，保留其中之一就可以了（只要代码完全相同）。事实上，大部分链接器也是这样做的。在链接的时候，链接器通过一些编译器辅助的手段将重复的模板函数代码 fun<int>(int) 删除掉，只保留了单个副本。这样一来，就解决了模板实例化时产生的代码冗余问题。我们可以看看图 2-1 中的模板函数的编译与链接的过程示意。

不过读者也注意到了，对于源代码中出现的每一处模板实例化，编译器都需要去做实例化的工作；而在链接时，链接器还需要移除重复的实例化代码。很明显，这样的工作太过冗余，而在广泛使用模板的项目中，由于编译器会产生大量冗余代码，会极大地增加编译器的编译时间和链接时间。解决这个问题的方法基本跟变量共享的思路是一样的，就是使用“外部的”模板。

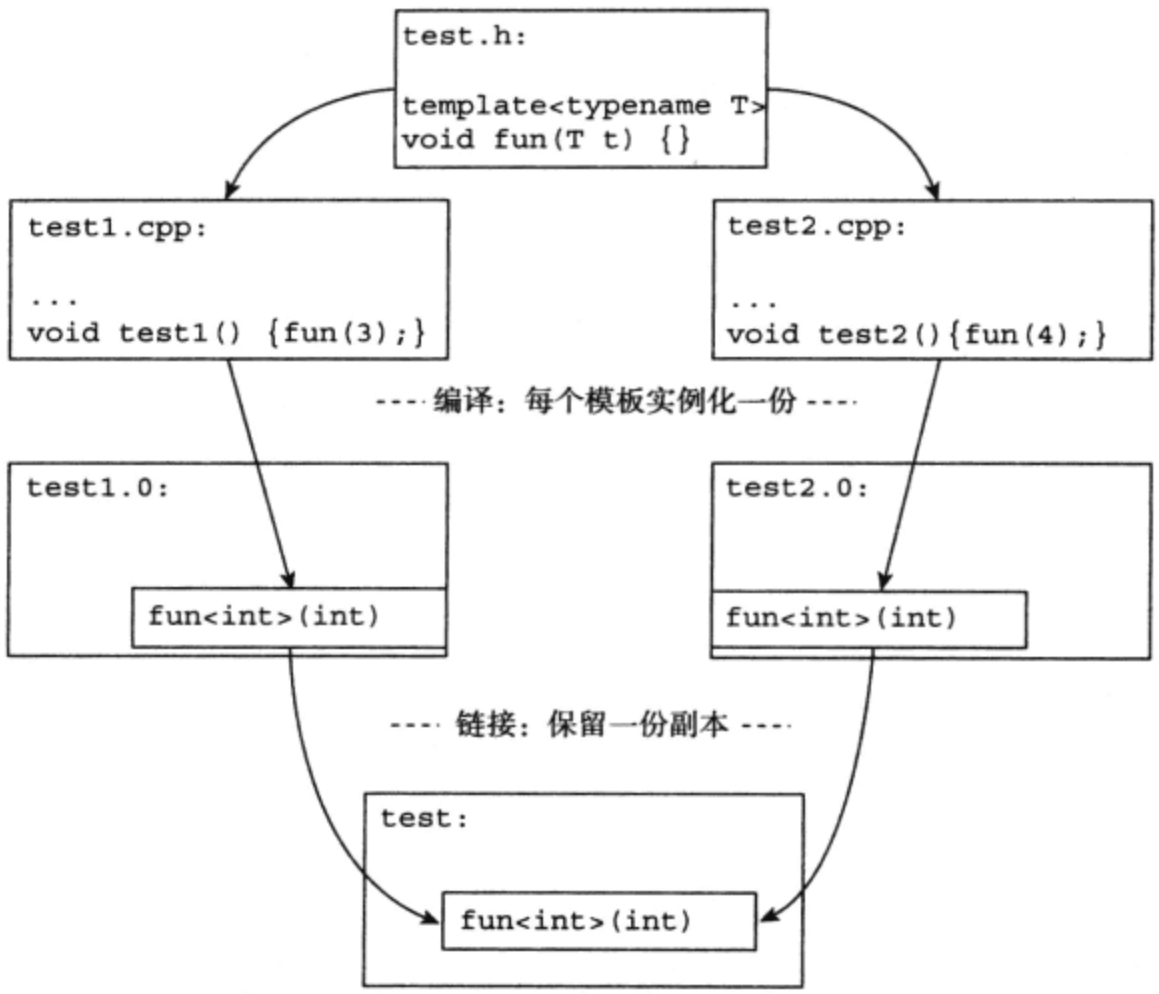


图 2-1 模板函数的编译与链接

2.12.2 显式的实例化与外部模板的声明

外部模板的使用实际依赖于C++98中一个已有的特性，即显式实例化（Explicit Instantiation）。显式实例化的语法很简单，比如对于以下模板：

```
template <typename T> void fun(T) {}
```

我们只需要声明：

```
template void fun<int>(int);
```

这就可以使编译器在本编译单元中实例化出一个 fun<int>(int) 版本的函数（这种做法也被称为强制实例化）。而在 C++11 标准中，又加入了外部模板（Extern Template）的声明。语法上，外部模板的声明跟显式的实例化差不多，只是多了一个关键字 extern。对于上面的例子，我们可以通过：

```
extern template void fun<int>(int);
```

这样的语法完成一个外部模板的声明。

那么回到一开始我们的例子，来修改一下我们的代码。首先，在 test1.cpp 做显式地实例化：

```
#include "test.h"
template void fun<int>(int); // 显示地实例化
void test1() { fun(3); }
```

接下来，在 test2.cpp 中做外部模板的声明：

```
#include "test.h"
extern template void fun<int>(int); // 外部模板的声明
void test1() { fun(3); }
```

这样一来，在 test2.o 中不会再生成 fun<int>(int) 的实例代码。整个模板的实例化流程如图 2-2 所示。

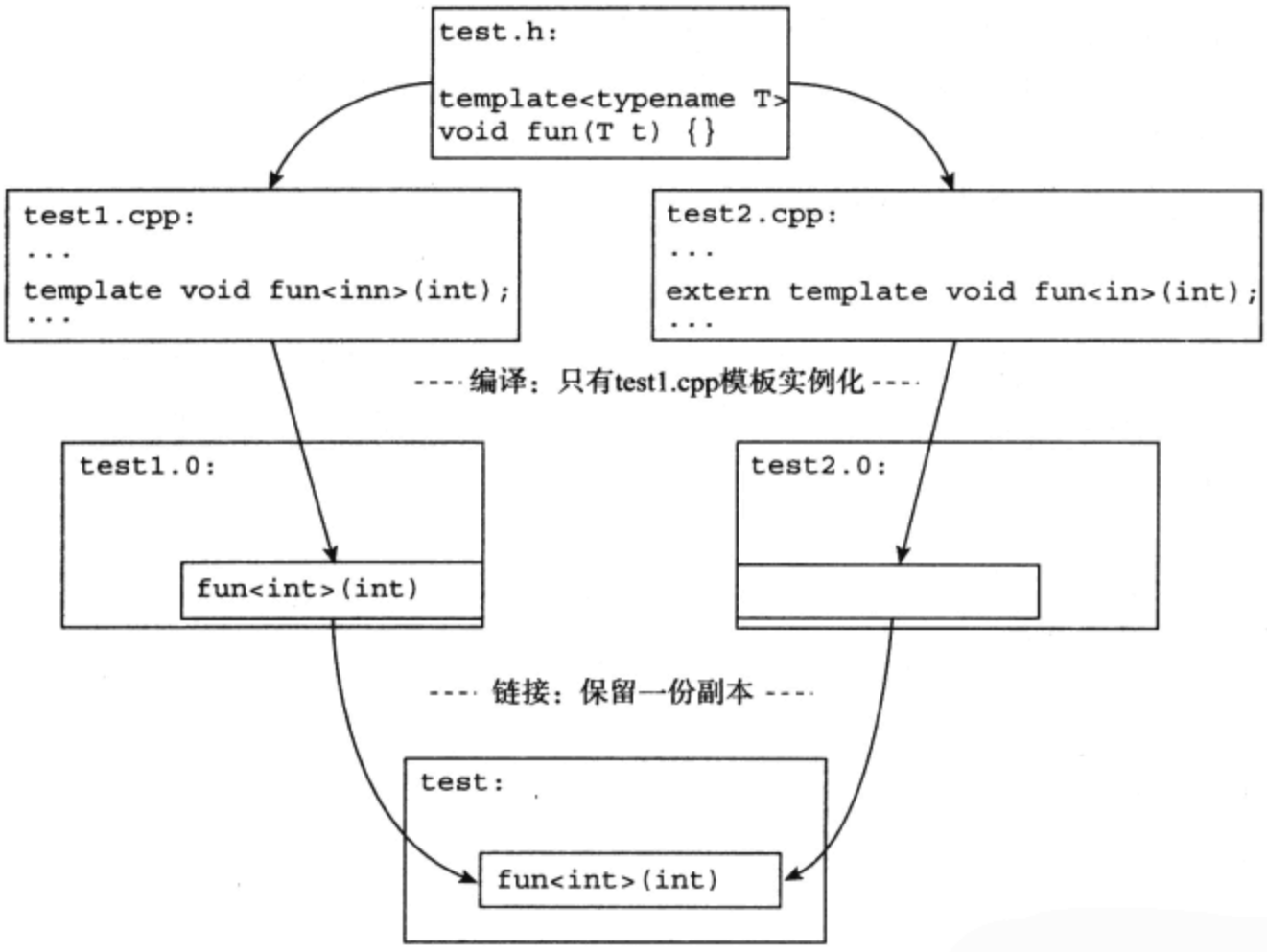


图 2-2 模板函数的编译与链接（使用外部模板声明）

可以看到，由于 test2.o 不再包含 fun<int>(int) 的实例，因此链接器的工作很轻松，基本跟外部变量的做法是一样的，即只需要保证让 test1.cpp 和 test2.cpp 共享一份代码位置即可。而同时，编译器也不用每次都产生一份 fun<int>(int) 的代码，所以可以减少编译时间。这里也可以把外部模板声明放在头文件中，这样所有包含 test.h 的头文件就可以共享这个外部模板声明了。这一点跟使用外部变量声明是完全一致的。

在使用外部模板的时候，我们还需要注意以下问题：如果外部模板声明出现于某个编译单元中，那么与之对应的显示实例化必须出现于另一个编译单元中或者同一个编译单元的后

续代码中；外部模板声明不能用于一个静态函数（即文件域函数），但可以用于类静态成员函数（这一点是显而易见的，因为静态函数没有外部链接属性，不可能在本编译单元之外出现）。

在实际上，C++11 中“模板的显式实例化定义、外部模板声明和使用”好比“全局变量的定义、外部声明和使用”方式的再次应用。不过相比于外部变量声明，不使用外部模板声明并不会导致任何问题。如我们在本节开始讲到的，外部模板定义更应该算作一种针对编译器的编译时间及空间的优化手段。很多时候，由于程序员低估了模板实例化展开的开销，因此大量的模板使用会在代码中产生大量的冗余。这种冗余，有的时候已经使得编译器和链接器力不从心。但这并不意味着程序员需要为四五十行的代码写很多显式模板声明及外部模板声明。只有在项目比较大的情况下。我们才建议用户进行这样的优化。总的来说，就是在既不忽视模板实例化产生的编译及链接开销的同时，也不要过分担心模板展开的开销。

2.13 局部和匿名类型作模板实参

☞ **类别：部分人**

在 C++98 中，标准对模板实参的类型还有一些限制。具体地讲，局部的类型和匿名的类型在 C++98 中都不能做模板类的实参。比如，如代码清单 2-30 所示的代码在 C++98 中很多都无法编译通过。

代码清单 2-30

```
template <typename T> class X {};
template <typename T> void TempFun(T t){};
struct A{} a;
struct {int i;}b;              // b 是匿名类型变量
typedef struct {int i;}B;      // B 是匿名类型

void Fun()
{
    struct C {} c;             // C 是局部类型

    X<A> x1;     // C++98 通过，C++11 通过
    X<B> x2;     // C++98 错误，C++11 通过
    X<C> x3;     // C++98 错误，C++11 通过
    TempFun(a); // C++98 通过，C++11 通过
    TempFun(b); // C++98 错误，C++11 通过
    TempFun(c); // C++98 错误，C++11 通过
}
// 编译选项 :g++ -std=c++11 2-13-1.cpp
```

在代码清单 2-30 中，我们定义了一个模板类 X 和一个模板函数 TempFun，然后分别用普通的全局结构体、匿名的全局结构体，以及局部的结构体作为参数传给模板。可以看到，

使用了局部的结构体 C 及变量 c，以及匿名的结构体 B 及变量 b 的模板类和模板函数，在 C++98 标准下都无法通过编译。而除了匿名的结构体之外，匿名的联合体以及枚举类型，在 C++98 标准下也都是无法做模板的实参的。如今看来这都是不必要的限制。所以在 C++11 中标准允许了以上类型做模板参数的做法，故而用支持 C++11 标准的编译器编译以上代码，代码清单 2-30 所示代码可以通过编译。

不过值得指出的是，虽然匿名类型可以被模板参数所接受了，但并不意味着以下写法可以被接受，如代码清单 2-31 所示。

代码清单 2-31

```
template <typename T> struct MyTemplate { };

int main() {
    MyTemplate<struct { int a; }> t; // 无法编译通过，匿名类型的声明不能在模板实参位置
    return 0;
}
// 编译选项 :g++ -std=c++11 2-13-2.cpp
```

在代码清单 2-31 中，我们把匿名的结构体直接声明在了模板实参的位置。这种做法非常直观，但却不符合 C/C++ 的语法。在 C/C++ 中，即使是匿名类型的声明，也需要独立的表达式语句。要使用匿名结构体作为模板参数，则可如同代码清单 2-30 一样对匿名结构体作别名。此外在第 4 章我们还会看到使用 C++11 独有的类型推导 decltype，也可以完成相同的功能。

2.14 本章小结

在本章中，我们可以看到 C++11 大大小小共 17 处改动。这 17 处改动，主要都是为保持 C++ 的稳定性以及兼容性而增加的。

比如为了兼容 C99，C++11 引入了 4 个 C99 的预定的宏、__func__ 预定义标识符、_Pragma 操作符、变长参数定义，以及宽窄字符连接等概念。这些都是错过了 C++98 标准，却进入了 C99 的一些标准，为了最大程度地兼容 C，C++ 将这些特性全都纳入 C++11。而由于标准的更新，C++11 也更新了 __cplusplus 宏的值，以表示新的标准的到来。而为了稳定性，C++11 不仅纳入了 C99 中的 long long 类型，还将扩展整型的规则预先定义好。这样一来，就保证了各个编译器扩展内置类型遵守统一的规则。此外，C++11 还将做法不一的静态断言做成了编译器级别的支持，以方便程序员使用。而通过抛弃 throw() 异常描述符和新增可以推导是否抛出异常的 noexcept 异常描述符，C++11 又对标准库大量代码做了改进。

在类方面，C++11 先是对非静态成员的初始化做了改进，同时允许 sizeof 直接作用于类的成员，再者 C++11 对 friend 的声明予以了一定扩展，以方便通过模板的方式指定某个类是

否是其他类或者函数的友元。而 final 和 override 两个关键字的引入，则又为对象编程增加了实用的功能。而在模板方面，C++11 则把默认模板参数的概念延伸到了模板函数上。而且局部类型和匿名类型也可以用做模板的实参。这两个约束的解除，使得模板的使用中需要记忆的规则又少了一些。而外部模板声明的引入，C++11 又为很看重编译性能的用户提供了一些优化编译时间和内存消耗的方法。

在读者读完并理解了这些特性之后，会发现它们几乎像是一台轰鸣作响的机器上的螺丝钉、润滑油、电线丝。C++ 标准委员会则通过这些小修小补，让 C++11 已有的特性看起来更加成熟，更加完美。在这一章里，虽然有的特性会带来一些“小欣喜”，但我们还看不到脱胎换骨、让人眼前一亮的新特性。不过这些零散的特性又确实非常重要，是 C++ 发展中必要的“维护”过程的必然结果。

不过如同我们讲到的，C++11 其实已经看起来像一门新的语言了。在接下来的几章中，我们会看到更多更“闪亮”的新特性。如果读者已经等不及了，那么请现在就翻开下一页。

第③章

通用为本，专用为末

C++11 的设计者总是希望从各种方案中抽象出更为通用的方法来构建新的特性。这意味着 C++11 中的新特性往往具有广泛的可用性，可以与其他已有的，或者新增的语言特性结合起来进行自由的组合，或者提升已有特性的通用性。这与在语言缺陷上“打补丁”的做法有着本质的不同，但也在一定程度上拖慢了 C++11 标准的制定。不过现在一切都已经尘埃落定了。在本章里读者可以看到这些经过反复斟酌制定的新特性，并体会其“普适”的特性。当然，要对一些形如右值引用、移动语义的复杂新特性做到融会贯通，则需要读者反复揣摩。

3.1 继承构造函数

☞ **类别：类作者**

C++ 中的自定义类型——类，是 C++ 面向对象的基石。类具有可派生性，派生类可以自动获得基类的成员变量和接口（虚函数和纯虚函数，这里我们指的都是 public 派生）。不过基类的非虚函数则无法再被派生类使用了。这条规则对于类中最为特别的构造函数也不例外，如果派生类要使用基类的构造函数，通常需要在构造函数中显式声明。比如下面的例子：

```
struct A { A(int i) {} };
struct B : A { B(int i): A(i) {} };
```

B 派生于 A，B 又在构造函数中调用 A 的构造函数，从而完成构造函数的“传递”。这在 C++ 代码中非常常见。当然，这样的设计有一定的好处，尤其是 B 中有成员的时候。如代码清单 3-1 所示的例子。

代码清单 3-1

```
struct A { A(int i) {} };
struct B : A {
    B(int i): A(i), d(i) {}
    int d;
};
// 编译选项 :g++ -c 3-1-1.cpp
```

在代码清单 3-1 中我们看到，派生于结构体 A 的结构体 B 拥有一个成员变量 d，那么在 B 的构造函数 B(int i) 中，我们可以在初始化其基类 A 的同时初始化成员 d。从这个意义上

讲，这样的构造函数设计也算是非常合理的。

不过合情合理并不等于合用，有的时候，我们的基类可能拥有数量众多的不同版本的构造函数——这样的情况并不少见，我们在 2.7 节中就曾经看到过这样的例子。那么倘若基类中有大量的构造函数，而派生类却只有一些成员函数时，那么对于派生类而言，其构造就等同于构造基类。这时候问题就来了，在派生类中我们写的构造函数完完全全就是为了构造基类。那么为了遵从于语法规则，我们还需要写很多的“透传”的构造函数。我们可以看看下面这个例子，如代码清单 3-2 所示。

代码清单 3-2

```
struct A {
    A(int i) {}
    A(double d, int i) {}
    A(float f, int i, const char* c) {}
    // ...
};

struct B : A {
    B(int i): A(i) {}
    B(double d, int i) : A(d, i) {}
    B(float f, int i, const char* c) : A(f, i, c){}
    // ...
    virtual void ExtraInterface(){}
};
// 编译选项 :g++ -c 3-1-2.cpp
```

在代码清单 3-2 中，我们的基类 A 有很多的构造函数的版本，而继承于 A 的派生类 B 实际上只是添加了一个接口 ExtraInterface。那么如果我们在构造 B 的时候想要拥有 A 这样多的构造方法的话，就必须一一“透传”各个接口。这无疑是相当不方便的。

事实上，在 C++ 中已经有了一个好用的规则，就是如果派生类要使用基类的成员函数的话，可以通过 using 声明（using-declaration）来完成。我们可以看看下面这个例子，如代码清单 3-3 所示。

代码清单 3-3

```
#include <iostream>
using namespace std;

struct Base {
    void f(double i){ cout << "Base:" << i << endl; }
};

struct Derived : Base {
    using Base::f;
```

```
    void f(int i) { cout << "Derived:" << i << endl; }
};

int main() {
    Base b;
    b.f(4.5);   // Base:4.5

    Derived d;
    d.f(4.5);   // Base:4.5
}
// 编译选项 :g++ 3-1-3.cpp
```

在代码清单 3-3 中，我们的基类 Base 和派生类 Derived 声明了同名的函数 f，不过在派生类中的版本跟基类有所不同。派生类中的 f 函数接受 int 类型为参数，而基类中接受 double 类型的参数。这里我们使用了 using 声明，声明派生类 Derived 也使用基类版本的函数 f。这样一来，派生类中实际就拥有了两个 f 函数的版本。可以看到，我们在 main 函数中分别定义了 Base 变量 b 和 Derived 变量 d，并传入浮点字面常量 4.5，结果都会调用到基类的接受 double 为参数的版本。

在 C++11 中，这个想法被扩展到了构造函数上。子类可以通过使用 using 声明来声明继承基类的构造函数。那我们要改造代码清单 3-2 所示的例子就非常容易了，如代码清单 3-4 所示。

代码清单 3-4

```
struct A {
    A(int i) {}
    A(double d, int i) {}
    A(float f, int i, const char* c) {}
    // ...
};

struct B : A {
    using A::A;      // 继承构造函数
    // ...
    virtual void ExtraInterface(){}
};
```

这里我们通过 using A::A 的声明，把基类中的构造函数悉数继承到派生类 B 中。这样我们在代码清单 3-2 中的“透传”构造函数就不再需要了。而且更为精巧的是，C++11 标准继承构造函数被设计为跟派生类中的各种类默认函数（默认构造、析构、拷贝构造等）一样，是隐式声明的。这意味着如果一个继承构造函数不被相关代码使用，编译器不会为其产生真正的函数代码。这无疑比“透传”方案总是生成派生类的各种构造函数更加节省目标代码空间。

不过继承构造函数只会初始化基类中成员变量，对于派生类中的成员变量，则无能为力。不过配合我们2.7节中的类成员的初始化表达式，为派生类成员变量设定一个默认值还是没有问题的。

在代码清单3-5中我们就同时使用了继承构造函数和成员变量初始化两个C++11的特性。这样就可以解决一些继承构造函数无法初始化的派生类成员问题。如果这样仍然无法满足需求的话，程序员只能自己来实现一个构造函数，以达到基类和成员变量都能够初始化的目的。

代码清单3-5

```
struct A {
    A(int i) {}
    A(double d, int i) {}
    A(float f, int i, const char* c) {}
    // ...
};

struct B : A {
    using A::A;
    int d {0};
};

int main() {
    B b(356);    // b.d被初始化为0
}
```

有的时候，基类构造函数的参数会有默认值。对于继承构造函数来讲，参数的默认值是不会被继承的。事实上，默认值会导致基类产生多个构造函数的版本，这些函数版本都会被派生类继承。比如代码清单3-6所示的这个例子。

代码清单3-6

```
struct A {
    A (int a = 3, double = 2.4){}
}

struct B : A{
    using A::A;
};
```

可以看到，在代码清单3-6中，我们的基类的构造函数A (int a = 3, double = 2.4)有一个接受两个参数的构造函数，且两个参数均有默认值。那么A到底有多少个可能的构造函数的版本呢?

事实上，B可能从A中继承来的候选继承构造函数有如下一些：

❑A(int = 3, double = 2.4); 这是使用两个参数的情况。
❑A(int = 3); 这是减掉一个参数的情况。
❑A(const A &); 这是默认的复制构造函数。
❑A(); 这是不使用参数的情况。

相应地，B 中的构造函数将会包括以下一些：

❑B(int, double); 这是一个继承构造函数。
❑B(int); 这是减少掉一个参数的继承构造函数。
❑B(const B &); 这是复制构造函数，这不是继承来的。
❑B(); 这是不包含参数的默认构造函数。

可以看见，参数默认值会导致多个构造函数版本的产生，因此程序员在使用有参数默认值的构造函数的基类的时候，必须小心。

而有的时候，我们还会遇到继承构造函数“冲突”的情况。这通常发生在派生类拥有多个基类的时候。多个基类中的部分构造函数可能导致派生类中的继承构造函数的函数名、参数（有的时候，我们也称其为函数签名）都相同，那么继承类中的冲突的继承构造函数将导致不合法的派生类代码，如代码清单 3-7 所示。

代码清单 3-7

```
struct A { A(int) {} };
struct B { B(int) {} };

struct C: A, B {
    using A::A;
    using B::B;
};
```

在代码清单 3-7 中，A 和 B 的构造函数会导致 C 中重复定义相同类型的继承构造函数。这种情况下，可以通过显式定义继承类的冲突的构造函数，阻止隐式生成相应的继承构造函数来解决冲突。比如：

```
struct C: A, B {
    using A::A;
    using B::B;
    C(int){}
};
```

其中的构造函数 C(int) 就很好地解决了代码清单 3-7 中继承构造函数的冲突问题。

另外我们还需要了解的一些规则是，如果基类的构造函数被声明为私有成员函数，或者派生类是从基类中虚继承的，那么就不能够在派生类中声明继承构造函数。此外，如果一旦使用了继承构造函数，编译器就不会再为派生类生成默认构造函数了，那么形如代码清单

3-8 中这样的情况，程序员就必须注意继承构造函数没有包含一个无参数的版本。在本例中，变量 b 的定义应该是不能够通过编译的。

代码清单 3-8

```
struct A { A (int){} };
struct B : A{ using A::A; };

B b;     // B 没有默认构造函数
```

在我们编写本书的时候，还没有编译器实现了继承构造函数这个特性，所以本节中代码清单 3-4 至代码清单 3-8 的例子都仅供读者参考，因为我们并没有实际编译过。但是编译器对继承构造函数的支持应该很快就要完成了，比如 g++ 就计划在 4.8 版本中提供支持。可能本书出版的时候，读者就已经可以进行实验了。

3.2 委派构造函数

☞ **类别：类作者**

与继承构造函数类似的，委派构造函数也是 C++11 中对 C++ 的构造函数的一项改进，其目的也是为了减少程序员书写构造函数的时间。通过委派其他构造函数，多构造函数的类编写将更加容易。

首先我们可以看看代码清单 3-9 中构造函数代码冗余的例子。

代码清单 3-9

```
class Info {
public:
    Info() : type(1), name('a') { InitRest(); }
    Info(int i) : type(i), name('a') { InitRest(); }
    Info(char e): type(1), name(e) { InitRest(); }

private:
    void InitRest() { /* 其他初始化 */ }
    int  type;
    char name;
    // ...
};
// 编译选项 :g++ -c 3-2-1.cpp
```

在代码清单 3-9 中，我们声明了一个 Info 的自定义类型。该类型拥有 2 个成员变量以及 3 个构造函数。这里的 3 个构造函数都声明了初始化列表来初始化成员 type 和 name，并且都调用了相同的函数 InitRest。可以看到，除了初始化列表有的不同，而其他的部分，3 个构造函数基本上是相似的，因此其代码存在着很多重复。

读者可能会想到2.7节中我们对成员初始化的方法，那么我们用该方法来改写一下这个例子，如代码清单3-10所示。

代码清单3-10

```
class Info {
public:
    Info() { InitRest(); }
    Info(int i) : type(i) { InitRest(); }
    Info(char e): name(e) { InitRest(); }

private:
    void InitRest() { /* 其他初始化 */ }
    int  type {1};
    char name {'a'};
    // ...
};
// 编译选项 :g++ -c -std=c++11 3-2-2.cpp
```

在代码清单3-10中，我们在Info成员变量type和name声明的时候就地进行了初始化。可以看到，构造函数确实简单了不少，不过每个构造函数还是需要调用InitRest函数进行初始化。而现实编程中，构造函数中的代码还会更长，比如可能还需要调用一些基类的构造函数等。那能不能在一些构造函数中连InitRest都不用调用呢？

答案是肯定的，但前提是我们能够将一个构造函数设定为“基准版本”，比如本例中Info()版本的构造函数，而其他构造函数可以通过委派“基准版本”来进行初始化。按照这个想法，我们可能会如下编写构造函数：

```
Info() { InitRest(); }
Info(int i) { this->Info(); type = i; }
Info(char e) { this->Info(); name = e; }
```

这里我们通过this指针调用我们的“基准版本”的构造函数。不过可惜的是，一般的编译器都会阻止this->Info()的编译。原则上，编译器不允许在构造函数中调用构造函数，即使参数看起来并不相同。

当然，我们还可以开发出一个更具有“黑客精神”的版本：

```
Info() { InitRest(); }
Info(int i) { new (this) Info(); type = i; }
Info(char e) { new (this) Info(); name = e; }
```

这里我们使用了placement new来强制在本对象地址（this指针所指地址）上调用类的构造函数。这样一来，我们可以绕过编译器的检查，从而在2个构造函数中调用我们的“基准版本”。这种方法看起来不错，却是在已经初始化一部分的对象上再次调用构造函数，因此虽然针对这个简单的例子在我们的实验机上该做法是有效的，却是种危险的做法。

在 C++11 中，我们可以使用委派构造函数来达到期望的效果。更具体的，C++11 中的委派构造函数是在构造函数的初始化列表位置进行构造的、委派的。我们可以看看代码清单 3-11 所示的这个例子。

代码清单 3-11

```
class Info {
public:
    Info() { InitRest(); }
    Info(int i) : Info() { type = i; }
    Info(char e): Info() { name = e; }

private:
    void InitRest() { /* 其他初始化 */ }
    int  type {1};
    char name {'a'};
    // ...
};
// 编译选项 :g++ -c -std=c++11 3-2-3.cpp
```

可以看到，在代码清单 3-11 中，我们在 Info(int) 和 Info(char) 的初始化列表的位置，调用了“基准版本”的构造函数 Info()。这里我们为了区分被调用者和调用者，称在初始化列表中调用“基准版本”的构造函数为委派构造函数（delegating constructor），而被调用的“基准版本”则为目标构造函数（target constructor）。在 C++11 中，所谓委派构造，就是指委派函数将构造的任务委派给了目标构造函数来完成这样一种类构造的方式。

当然，在代码清单 3-11 中，委派构造函数只能在函数体中为 type、name 等成员赋初值。这是由于委派构造函数不能有初始化列表造成的。在 C++ 中，构造函数不能同时“委派”和使用初始化列表，所以如果委派构造函数要给变量赋初值，初始化代码必须放在函数体中。比如：

```
struct Rule1 {
    int i;
    Rule1(int a): i(a) {}
    Rule1(): Rule1(40), i(1) {} // 无法通过编译
};
```

Rule1 的委派构造函数 Rule1() 的写法就是非法的。我们不能在初始化列表中既初始化成员，又委托其他构造函数完成构造。

这样一来，代码清单 3-11 中的代码的初始化就不那么令人满意了，因为初始化列表的初始化方式总是先于构造函数完成的（实际在编译完成时就已经决定了）。这会可能致使程序员犯错（稍后解释）。不过我们可以稍微改造一下目标构造函数，使得委派构造函数依然可以在初始化列表中初始化所有成员，如代码清单 3-12 所示。

代码清单 3-12

```
class Info {
public:
    Info() : Info(1, 'a') { }
    Info(int i) : Info(i, 'a') { }
    Info(char e): Info(1, e) { }

private:
    Info(int i, char e): type(i), name(e) { /* 其他初始化 */ }
    int  type;
    char name;
    // ...
};
// 编译选项 :g++ -c -std=c++11 3-2-4.cpp
```

在代码清单 3-12 中，我们定义了一个私有的目标构造函数 Info(int, char)，这个构造函数接受两个参数，并将参数在初始化列表中初始化。而且由于这个目标构造函数的存在，我们可以不再需要 InitRest 函数了，而是将其代码都放入 Info(int, char) 中。这样一来，其他委派构造函数就可以委托该目标构造函数来完成构造。

事实上，在使用委派构造函数的时候，我们也建议程序员抽象出最为“通用”的行为做目标构造函数。这样做一来代码清晰，二来行为也更加正确。读者可以比较一下代码清单 3-11 和代码清单 3-12 中 Info 的定义，这里我们假设代码清单 3-11、代码清单 3-12 中注释行的“其他初始化”位置的代码如下：

```
type += 1;
```

那么调用 Info(int) 版本的构造函数会得到不同的结果。比如如果做如下一个类型的声明：

```
Info f(3);
```

这个声明对代码清单 3-11 中的 Info 定义而言，会导致成员 f.type 的值为 3，（因为 Info(int) 委托 Info() 初始化，后者调用 InitRest 将使得 type 的值为 4。不过 Info(int) 函数体内又将 type 重写为 3 ）。而依照代码清单 3-12 中的 Info 定义，f.type 的值将最终为 4。从代码编写者角度看，代码清单 3-12 中 Info 的行为会更加正确。这是由于在 C++11 中，目标构造函数的执行总是先于委派构造函数而造成的。因此避免目标构造函数和委托构造函数体中初始化同样的成员通常是必要的，否则则可能发生代码清单 3-11 错误。

而在构造函数比较多的时候，我们可能会拥有不止一个委派构造函数，而一些目标构造函数很可能也是委派构造函数，这样一来，我们就可能在委派构造函数中形成链状的委派构造关系，如代码清单 3-13 所示。

代码清单 3-13

```
class Info {
public:
    Info() : Info(1) { }     // 委派构造函数
    Info(int i) : Info(i, 'a') { } // 既是目标构造函数，也是委派构造函数
    Info(char e): Info(1, e) { }

private:
    Info(int i, char e): type(i), name(e) { /* 其他初始化 */ } // 目标构造函数
    int  type;
    char name;
    // ...
};
// 编译选项 :g++ -c -std=c++11 3-2-5.cpp
```

代码清单 3-13 所示就是这样一种链状委托构造，这里我们使 Info() 委托 Info(int) 进行构造，而 Info(int) 又委托 Info(int, char) 进行构造。在委托构造的链状关系中，有一点程序员必须注意，就是不能形成委托环（delegation cycle）。比如：

```
struct Rule2 {
    int i, c;
    Rule2(): Rule2(2) {}
    Rule2(int i): Rule2('c') {}
    Rule2(char c): Rule2(2) {}
};
```

Rule2 定义中，Rule2()、Rule2(int) 和 Rule2(char) 都依赖于别的构造函数，形成环委托构造关系。这样的代码通常会导致编译错误。

委派构造的一个很实际的应用就是使用构造模板函数产生目标构造函数，如代码清单 3-14 所示。

代码清单 3-14

```
#include <list>
#include <vector>
#include <deque>
using namespace std;

class TDConstructed {
    template<class T> TDConstructed(T first, T last) :
        l(first, last) {}
    list<int> l;

public:
    TDConstructed(vector<short> & v):
        TDConstructed(v.begin(), v.end()) {}
    TDConstructed(deque<int> & d):
```

```
        TDConstructed(d.begin(), d.end()) {}
};
// 编译选项 :g++ -c -std=c++11 3-2-6.cpp
```

在代码清单 3-14 中，我们定义了一个构造函数模板。而通过两个委派构造函数的委托，构造函数模板会被实例化。T 会分别被推导为 vector<short>::iterator 和 deque<int>::iterator 两种类型。这样一来，我们的 TDConstructed 类就可以很容易地接受多种容器对其进行初始化。这无疑比罗列不同类型的构造函数方便了很多。可以说，委托构造使得构造函数的泛型编程也成为了一种可能。

此外，在异常处理方面，如果在委派构造函数中使用 try 的话，那么从目标构造函数中产生的异常，都可以在委派构造函数中被捕捉到。我们可以看看代码清单 3-15 所示的例子。

代码清单 3-15

```
#include <iostream>
using namespace std;

class DCExcept {
public:
    DCExcept(double d)
        try : DCExcept(1, d) {
            cout << "Run the body." << endl;
            // 其他初始化
            }
            catch(...) {
                cout << "caught exception." << endl;
            }
private:
    DCExcept(int i, double d){
        cout << "going to throw!" << endl;
        throw 0;
    }
    int type;
    double data;
};

int main() {
    DCExcept a(1.2);
}
// 编译选项 :g++ -std=c++11 3-2-7.cpp
```

在代码清单 3-15 中，我们在目标构造函数 DCExcept(int, double) 抛出了一个异常，并在委派构造函数 DCExcept(int) 中进行捕捉。编译运行该程序，我们在实验机上获得以下输出：

```
going to throw!
caught exception.
```

```
terminate called after throwing an instance of 'int'
Aborted
```

可以看到，由于在目标构造函数中抛出了异常，委派构造函数的函数体部分的代码并没有被执行。这样的设计是合理的，因为如果函数体依赖于目标构造函数构造的结果，那么当目标构造函数构造发生异常的情况下，还是不要执行委派构造函数函数体中的代码为好。

其实，在 Java 等一些面向对象的编程语言中，早已经支持了委派构造函数这样的功能。因此，相比于继承构造函数，委派构造函数的设计和实现都比较早。而通过成员的初始化、委派构造函数，以及继承构造函数，C++ 中的构造函数的书写将进一步简化，这对程序员尤其是库的编写者来说，无疑是有积极意义的。

3.3 右值引用：移动语义和完美转发

☞ 类别：类作者

3.3.1 指针成员与拷贝构造

对 C++ 程序员来说，编写 C++ 程序有一条必须注意的规则，就是在类中包含了一个指针成员的话，那么就要特别小心拷贝构造函数的编写，因为一不小心，就会出现内存泄露。我们来看看代码清单 3-16 中的例子。

代码清单 3-16

```
#include <iostream>
using namespace std;

class HasPtrMem {
public:
    HasPtrMem(): d(new int(0)) {}
    HasPtrMem(const HasPtrMem & h):
        d(new int(*h.d)) {} // 拷贝构造函数，从堆中分配内存，并用 *h.d 初始化
    ~HasPtrMem() { delete d; }
    int * d;
};

int main() {
    HasPtrMem a;
    HasPtrMem b(a);
    cout << *a.d << endl;    // 0
    cout << *b.d << endl;    // 0
}   // 正常析构
// 编译选项 :g++ 3-3-1.cpp
```

在代码清单 3-16 中，我们定义了一个 HasPtrMem 的类。这个类包含一个指针成员，该

成员在构造时接受一个 new 操作分配堆内存返回的指针，而在析构的时候则会被 delete 操作用于释放之前分配的堆内存。在 main 函数中，我们声明了 HasPtrMem 类型的变量 a，又使用 a 初始化了变量 b。按照 C++ 的语法，这会调用 HasPtrMem 的拷贝构造函数。这里的拷贝构造函数由编译器隐式生成，其作用是执行类似于 memcpy 的按位拷贝。这样的构造方式有一个问题，就是 a.d 和 b.d 都指向了同一块堆内存。因此在 main 作用域结束的时候，a 和 b 的析构函数纷纷被调用，当其中之一完成析构之后（比如 b），那么 a.d 就成了一个“悬挂指针”（dangling pointer），因为其不再指向有效的内存了。那么在该悬挂指针上释放内存就会造成严重的错误。

这个问题在 C++ 编程中非常经典。这样的拷贝构造方式，在 C++ 中也常被称为“浅拷贝”（shollow copy）。而在未声明构造函数的情况下，C++ 也会为类生成一个浅拷贝的构造函数。通常最佳的解决方案是用户自定义拷贝构造函数来实现“深拷贝”（deep copy），我们来看看代码清单 3-17 中的修正方法。

代码清单 3-17

```
#include <iostream>
using namespace std;

class HasPtrMem {
public:
    HasPtrMem(): d(new int(0)) {}
    HasPtrMem(HasPtrMem & h):
        d(new int(*h.d)) {} // 拷贝构造函数，从堆中分配内存，并用 *h.d 初始化
    ~HasPtrMem() { delete d; }
    int * d;
};

int main() {
    HasPtrMem a;
    HasPtrMem b(a);
    cout << *a.d << endl;    // 0
    cout << *b.d << endl;    // 0
}   // 正常析构
// 编译选项 :g++ 3-3-2.cpp
```

在代码清单 3-17 中，我们为 HasPtrMem 添加了一个拷贝构造函数。拷贝构造函数从堆中分配新内存，将该分配来的内存的指针交还给 d，又使用 *(h.d) 对 *d 进行了初始化。通过这样的方法，就避免了悬挂指针的困扰。

3.3.2 移动语义

拷贝构造函数中为指针成员分配新的内存再进行内容拷贝的做法在 C++ 编程中几乎被视为是不可违背的。不过在一些时候，我们确实不需要这样的拷贝构造语义。我们可以看看代

码清单 3-18 所示的例子。

代码清单 3-18

```
#include <iostream>
using namespace std;

class HasPtrMem {
public:
    HasPtrMem(): d(new int(0)) {
        cout << "Construct: " << ++n_cstr << endl;
    }
    HasPtrMem(const HasPtrMem & h): d(new int(*h.d)) {
        cout << "Copy construct: " << ++n_cptr << endl;
    }
    ~HasPtrMem() {
        cout << "Destruct: " << ++n_dstr << endl;
    }
    int * d;
    static int n_cstr;
    static int n_dstr;
    static int n_cptr;
};

int HasPtrMem::n_cstr = 0;
int HasPtrMem::n_dstr = 0;
int HasPtrMem::n_cptr = 0;

HasPtrMem GetTemp() { return HasPtrMem(); }

int main() {
    HasPtrMem a = GetTemp();
}
// 编译选项 :g++ 3-3-3.cpp -fno-elide-constructors
```

在代码清单 3-18 中，我们声明了一个返回一个 HasPtrMem 变量的函数。为了记录构造函数、拷贝构造函数，以及析构函数调用的次数，我们使用了一些静态变量。在 main 函数中，我们简单地声明了一个 HasPtrMem 的变量 a，要求它使用 GetTemp 的返回值进行初始化。编译运行该程序，我们可以看到下面的输出：

```
Construct: 1
Copy construct: 1
Destruct: 1
Copy construct: 2
Destruct: 2
Destruct: 3
```

这里构造函数被调用了一次，这是在 GetTemp 函数中 HasPtrMem() 表达式显式地调用了

构造函数而打印出来的。而拷贝构造函数则被调用了两次。这两次一次是从 GetTemp 函数中 HasPtrMem() 生成的变量上拷贝构造出一个临时值，以用作 GetTemp 的返回值，而另外一次则是由临时值构造出 main 中变量 a 调用的。对应地，析构函数也就调用了 3 次。这个过程如图 3-1 所示。

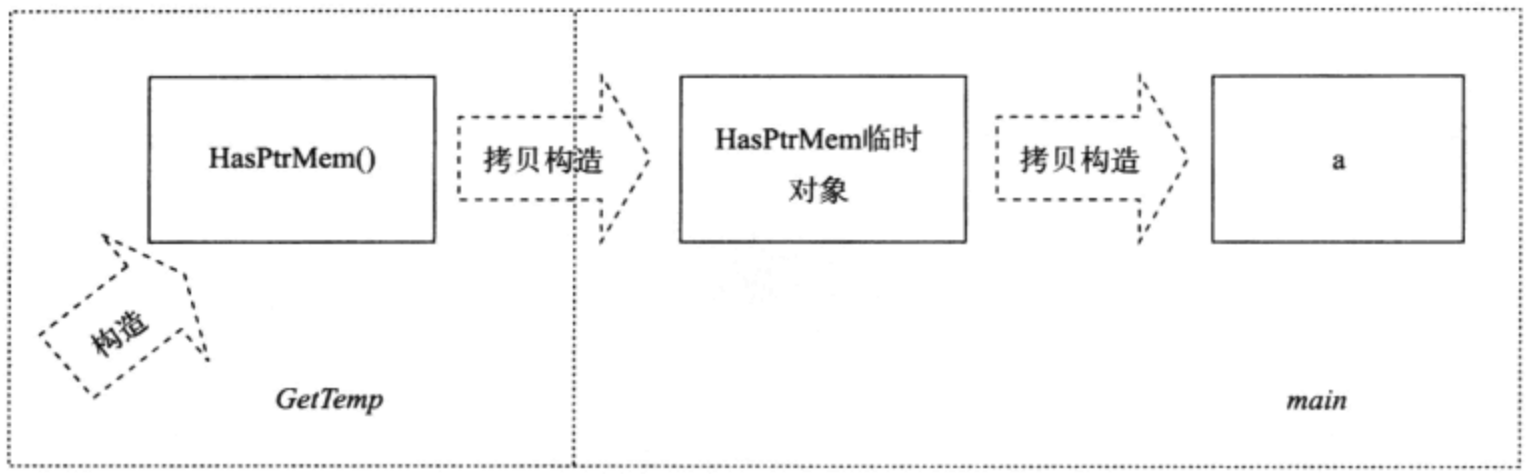


图 3-1　函数返回时的临时变量与拷贝

最让人感到不安就是拷贝构造函数的调用。在我们的例子里，类 HasPtrMem 只有一个 int 类型的指针。而如果 HasPtrMem 的指针指向非常大的堆内存数据的话，那么拷贝构造的过程就会非常昂贵。可以想象，这种情况一旦发生，a 的初始化表达式的执行速度将相当堪忧。而更为令人堪忧的是，临时变量的产生和销毁以及拷贝的发生对于程序员来说基本上是透明的，不会影响程序的正确性，因而即使该问题导致程序的性能不如预期，也不易被程序员察觉（事实上，编译器常常对函数返回值有专门的优化，我们在本节结束时会提到）。

让我们把目光再次聚集在临时对象上，即图 3-1 中的 main 函数的部分。按照 C++ 的语义，临时对象将在语句结束后被析构，会释放它所包含的堆内存资源。而 a 在拷贝构造的时候，又会被分配堆内存。这样的一去一来似乎并没有太大的意义，那么我们是否可以在临时对象构造 a 的时候不分配内存，即不使用所谓的拷贝构造语义呢？

在 C++11 中，答案是肯定的。我们可以看看如图 3-2 所示的示意图。

图 3-2 中的上半部分可以看到从临时变量中拷贝构造变量 a 的做法，即在拷贝时分配新的堆内存，并从临时对象的堆内存中拷贝内容至 a.d。而构造完成后，临时对象将析构，因此其拥有的堆内存资源会被析构函数释放。而图 3-2 的下半部分则是一种“新”方法（实际跟我们在代码清单 3-1 中做得差不多），该方法在构造时使得 a.d 指向临时对象的堆内存资源。同时我们保证临时对象不释放所指向的堆内存（下面解释怎么做），那么在构造完成后，临时对象被析构，a 就从中“偷”到了临时对象所拥有的堆内存资源。

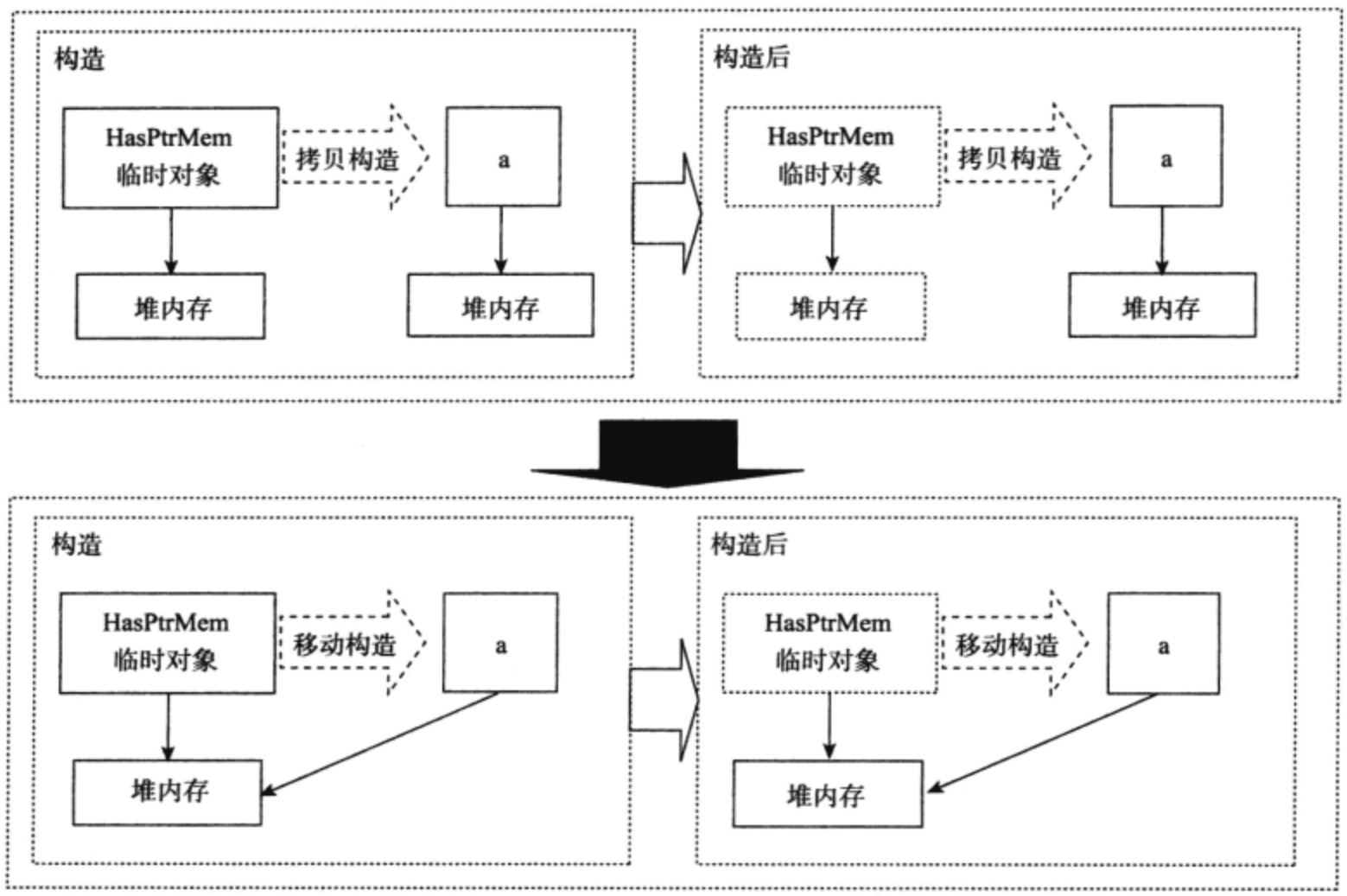


图 3-2 拷贝构造与移动构造

在 C++11 中，这样的“偷走”临时变量中资源的构造函数，就被称为“移动构造函数”。而这样的“偷”的行为，则称之为“移动语义”（move semantics）。当然，换成白话的中文，可以理解为“移为己用”。我们可以看看代码清单 3-19 中是如何来实现这种移动语义的。

代码清单 3-19

```
#include <iostream>
using namespace std;

class HasPtrMem {
public:
    HasPtrMem(): d(new int(3)) {
        cout << "Construct: " << ++n_cstr << endl;
    }
    HasPtrMem(const HasPtrMem & h): d(new int(*h.d)) {
        cout << "Copy construct: " << ++n_cptr << endl;
    }
    HasPtrMem(HasPtrMem && h): d(h.d) { // 移动构造函数
        h.d = nullptr;                  // 将临时值的指针成员置空
        cout << "Move construct: " << ++n_mvtr << endl;
    }
```

```
    ~HasPtrMem() {
        delete d;
        cout << "Destruct: " << ++n_dstr << endl;
    }
    int * d;
    static int n_cstr;
    static int n_dstr;
    static int n_cptr;
    static int n_mvtr;
};

int HasPtrMem::n_cstr = 0;
int HasPtrMem::n_dstr = 0;
int HasPtrMem::n_cptr = 0;
int HasPtrMem::n_mvtr = 0;

HasPtrMem GetTemp() {
    HasPtrMem h;
    cout << "Resource from " <<  __func__ << ": " << hex << h.d << endl;
    return h;
}

int main() {
    HasPtrMem a = GetTemp();
    cout << "Resource from " <<  __func__ << ": " << hex << a.d << endl;
}
// 编译选项 :g++ -std=c++11 3-3-4.cpp -fno-elide-constructors
```

相比于代码清单 3-18，代码清单 3-19 中的 HasPtrMem 类多了一个构造函数 HasPtrMem (HasPtrMem &&)，这个就是我们所谓的移动构造函数。与拷贝构造函数不同的是，移动构造函数接受一个所谓的“右值引用”的参数，关于右值我们接下来会解释，读者可以暂时理解为临时变量的引用。可以看到，移动构造函数使用了参数 h 的成员 d 初始化了本对象的成员 d（而不是像拷贝构造函数一样需要分配内存，然后将内容依次拷贝到新分配的内存中），而 h 的成员 d 随后被置为指针空值 nullptr（请参见 7.1 节，这里等同于 NULL）。这就完成了移动构造的全过程。

这里所谓的“偷”堆内存，就是指将本对象 d 指向 h.d 所指的内存这一条语句，相应地，我们还将 h 的成员 d 置为指针空值。这其实也是我们“偷”内存时必须做的。这是因为在移动构造完成之后，临时对象会立即被析构。如果不改变 h.d（临时对象的指针成员）的话，则临时对象会析构掉本是我们“偷”来的堆内存。这样一来，本对象中的 d 指针也就成了一个悬挂指针，如果我们对指针进行解引用，就会发生严重的运行时错误。

为了看看移动构造的效果，我们让 GetTemp 和 main 函数分别打印变量 h 和变量 a 中的指针 h.d 和 a.d，在我们的实验机上运行的结果如下：

```
Construct: 1
```

```
Resource from GetTemp: 0x603010
Move construct: 1
Destruct: 1
Move construct: 2
Destruct: 2
Resource from main: 0x603010
Destruct: 3
```

可以看到，这里没有调用拷贝构造函数，而是调用了两次移动构造函数，移动构造的结果是，GetTemp 中的 h 的指针成员 h.d 和 main 函数中的 a 的指针成员 a.d 的值是相同的，即 h.d 和 a.d 都指向了相同的堆地址内存。该堆内存在函数返回的过程中，成功地逃避了被析构的“厄运”，取而代之地，成为了赋值表达式中的变量 a 的资源。如果堆内存不是一个 int 长度的数据，而是以 MByte 为单位的堆空间，那么这样的移动带来的性能提升将非常惊人。

或许读者会质疑说：为什么要这么费力地添加移动构造函数呢？完全可以选择改变 GetTemp 的接口，比如直接传一个引用或者指针到 GetTemp 的参数中去，效果应该也不差。其实从性能上来讲，这样的做法确实毫无问题，甚至只好不差。不过从使用的方便性上来讲效果不好。如果函数返回临时值的话，可以在单条语句里完成很多计算，比如我们可以很自然地写出如下语句：

```
Caculate(GetTemp(), SomeOther(Maybe(), Useful(Values, 2)));
```

但如果通过传引用或者指针的方法而不返回值的话，通常就需要很多语句来完成上面的工作。可能是像下面这样的代码：

```
string *a; vector b;     // 事先声明一些变量用于传递返回值
...
Useful(Values, 2, a);     // 最后一个参数是指针，用于返回结果
SomeOther(Maybe(), a, b);     // 最后一个参数是引用，用于返回结果
Caculate(GetTemp(), b);
```

两者在代码编写效率和可读性上都存在着明显的差别。而即使声明这些传递返回值的变量为全局的，函数再将这些引用和指针都作为返回值返回给调用者，我们也需要在 Caculate 调用之前声明好所有的引用和指针。这无疑是繁琐的工作。函数返回临时变量的好处就是不需要声明变量，也不需要知道生命期。程序员只需要按照最自然的方式，使用最简单的语句就可以完成大量的工作。

那么再回到移动语义上来，还有一个最为关键的问题没有解决，那就是移动构造函数何时会被触发。之前我们只是提到了临时对象，一旦我们用到的是个临时变量，那么移动构造语义就可以得到执行。那么，在 C++ 中如何判断产生了临时对象？如何将其用于移动构造函数？是否只有临时变量可以用于移动构造？……读者可能还有很多问题。要回答这些问题，需要先了解一下 C++ 中的“值”是如何分类的。

注意 事实上，移动语义并不是什么新的概念，在 C++98/03 的语言和库中，它已经存

在了，比如：

- 在某些情况下拷贝构造函数的省略（copy constructor elision in some contexts）
- 智能指针的拷贝（auto_ptr“copy”）
- 链表拼接（list::splice）
- 容器内的置换（swap on containers）

以上这些操作都包含了从一个对象向另外一个对象的资源转移（至少概念上）的过程，唯一欠缺的是统一的语法和语义的支持，来使我们可以使用通用的代码移动任意的对象（就像我们今天可以使用通用的代码来拷贝任意对象一样）。如果能够任意地使用对象的移动而不是拷贝，那么标准库中的很多地方的性能都会大大提高。

3.3.3 左值、右值与右值引用

在C语言中，我们常常会提起左值（lvalue）、右值（rvalue）这样的称呼。而在编译程序时，编译器有时也会在报出的错误信息中会包含左值、右值的说法。不过左值、右值通常不是通过一个严谨的定义而为人所知的，大多数时候左右值的定义与其判别方法是一体的。一个最为典型的判别方法就是，在赋值表达式中，出现在等号左边的就是“左值”，而在等号右边的，则称为“右值”。比如：

```
a = b + c;
```

在这个赋值表达式中，a就是一个左值，而b + c则是一个右值。这种识别左值、右值的方法在C++中依然有效。不过C++中还有一个被广泛认同的说法，那就是可以取地址的、有名字的就是左值，反之，不能取地址的、没有名字的就是右值。那么这个加法赋值表达式中，&a是允许的操作，但&(b + c)这样的操作则不会通过编译。因此a是一个左值，(b + c)是一个右值。

这些判别方法通常都非常有效。更为细致地，在C++11中，右值是由两个概念构成的，一个是将亡值（xvalue，eXpiring Value），另一个则是纯右值（prvalue，Pure Rvalue）。

其中纯右值就是C++98标准中右值的概念，讲的是用于辨识临时变量和一些不跟对象关联的值。比如非引用返回的函数返回的临时变量值（我们在前面多次提到了）就是一个纯右值。一些运算表达式，比如1 + 3产生的临时变量值，也是纯右值。而不跟对象关联的字面量值，比如：2、‘c’、true，也是纯右值。此外，类型转换函数的返回值、lambda表达式（见7.3节）等，也都是右值。

而将亡值则是C++11新增的跟右值引用相关的表达式，这样表达式通常是将要被移动的对象（移为他用），比如返回右值引用T&&的函数返回值、std::move的返回值（稍后解释），或者转换为T&&的类型转换函数的返回值（稍后解释）。而剩余的，可以标识函数、对象的值都属于左值。在C++11的程序中，所有的值必属于左值、将亡值、纯右值三者之一。

注意 事实上，之所以我们只知道一些关于左值、右值的判断而很少听到其真正的定义的一个原因就是——很难归纳。而且即使归纳了，也需要大量的解释。

在 C++11 中，右值引用就是对一个右值进行引用的类型。事实上，由于右值通常不具有名字，我们也只能通过引用的方式找到它的存在。通常情况下，我们只能是从右值表达式获得其引用。比如：

```
T && a = ReturnRvalue();
```

这个表达式中，假设 ReturnRvalue 返回一个右值，我们就声明了一个名为 a 的右值引用，其值等于 ReturnRvalue 函数返回的临时变量的值。

为了区别于 C++98 中的引用类型，我们称 C++98 中的引用为“左值引用”（lvalue reference）。右值引用和左值引用都是属于引用类型。无论是声明一个左值引用还是右值引用，都必须立即进行初始化。而其原因可以理解为是引用类型本身自己并不拥有所绑定对象的内存，只是该对象的一个别名。左值引用是具名变量值的别名，而右值引用则是不具名（匿名）变量的别名。

在上面的例子中，ReturnRvalue 函数返回的右值在表达式语句结束后，其生命也就终结了（通常我们也称其具有表达式生命期），而通过右值引用的声明，该右值又“重获新生”，其生命期将与右值引用类型变量 a 的生命期一样。只要 a 还“活着”，该右值临时量将会一直“存活”下去。

所以相比于以下语句的声明方式：

```
T b = ReturnRvalue();
```

我们刚才的右值引用变量声明，就会少一次对象的析构及一次对象的构造。因为 a 是右值引用，直接绑定了 ReturnRvalue() 返回的临时量，而 b 只是由临时值构造而成的，而临时量在表达式结束后会析构因应就会多一次析构和构造的开销。

不过值得指出的是，能够声明右值引用 a 的前提是 ReturnRvalue 返回的是一个右值。通常情况下，右值引用是不能够绑定到任何的左值的。比如下面的表达式就是无法通过编译的。

```
int c;
int && d = c;
```

相对地，在 C++98 标准中就已经出现的左值引用是否可以绑定到右值（由右值进行初始化）呢？比如：

```
T & e = ReturnRvalue();
const T & f = ReturnRvalue();
```

这样的语句是否能够通过编译呢？这里的答案是：e 的初始化会导致编译时错误，而 f 则不会。

出现这样的状况的原因是，在常量左值引用在 C++98 标准中开始就是个“万能”的引用类型。它可以接受非常量左值、常量左值、右值对其进行初始化。而且在使用右值对其初始化的时候，常量左值引用还可以像右值引用一样将右值的生命期延长。不过相比于右值引用所引用的右值，常量左值所引用的右值在它的“余生”中只能是只读的。相对地，非常量左值只能接受非常量左值对其进行初始化。

既然常量左值引用在 C++98 中就已经出现，读者可能会努力地搜索记忆，想找出在 C++ 中使用常量左值绑定右值的情况。不过可能一切并不如愿。这是因为，在 C++11 之前，左值、右值对于程序员来说，一直呈透明状态。不知道什么是左值、右值，并不影响写出正确的 C++ 代码。引用的是左值和右值通常也并不重要。不过事实上，在 C++98 通过左值引用来绑定一个右值的情况并不少见，比如：

```
const bool & judgement = true;
```

就是一个使用常量左值引用来绑定右值的例子。不过与如下声明相比较看起来似乎差别不大。

```
const bool judgement = true;
```

可能很多程序员都没有注意到其中的差别（从语法上讲，前者直接使用了右值并为其“续命”，而后者的右值在表达式结束后就销毁了）。

事实上，即使在 C++98 中，我们也常可以使用常量左值引用来减少临时对象的开销，如代码清单 3-20 所示。

代码清单 3-20

```
#include <iostream>
using namespace std;

struct Copyable {
    Copyable() {}
    Copyable(const Copyable &o) {
        cout << "Copied" << endl;
    }
};

Copyable ReturnRvalue() { return Copyable(); }
void AcceptVal(Copyable) {}
void AcceptRef(const Copyable & ) {}

int main() {
    cout << "Pass by value: " << endl;
    AcceptVal(ReturnRvalue()); // 临时值被拷贝传入
    cout << "Pass by reference: " << endl;
    AcceptRef(ReturnRvalue()); // 临时值被作为引用传递
}
```

```
// 编译选项 :g++ 3-3-5.cpp -fno-elide-constructors
```

在代码清单 3-20 中，我们声明了结构体 Copyable，该结构体的唯一的作用就是在被拷贝构造的时候打印一句话：Copied。而两个函数，AcceptVal 使用了值传递参数，而 AcceptRef 使用了引用传递。在以 ReturnRvalue 返回的右值为参数的时候，AcceptRef 就可以直接使用产生的临时值（并延长其生命期），而 AcceptVal 则不能直接使用临时对象。

编译运行代码清单 3-20，可以得到以下结果：

```
Pass by value:
Copied
Copied
Pass by reference:
Copied
```

可以看到，由于使用了左值引用，临时对象被直接作为函数的参数，而不需要从中拷贝一次。读者可以自行分析一下输出结果，这里就不赘述了。而在 C++11 中，同样地，如果在代码清单 3-20 中以右值引用为参数声明如下函数：

```
void AcceptRvalueRef(Copyable && ) {}
```

也同样可以减少临时变量拷贝的开销。进一步地，还可以在 AcceptRvalueRef 中修改该临时值（这个时候临时值由于被右值引用参数所引用，已经获得了函数时间的生命期）。不过修改一个临时值的意义通常不大，除非像 3.3.2 节一样使用移动语义。

就本例而言，如果我们这样实现函数：

```
void AcceptRvalueRef(Copyable && s) {
   Copyable news = std::move(s);
}
```

这里 std::move 的作用是强制一个左值成为右值（看起来很奇怪？这个我们会在下面一节中解释）。该函数就是使用右值来初始化 Copyable 变量 news。当然，如同我们在上小节提到的，使用移动语义的前提是 Copyable 还需要添加一个以右值引用为参数的移动构造函数，比如：

```
Copyable(Copyable &&o) { /* 实现移动语义 */ }
```

这样一来，如果 Copyable 类的临时对象（即 ReturnRvalue 返回的临时值）中包含一些大块内存的指针，news 就可以如同代码清单 3-19 一样将临时值中的内存“窃”为己用，从而从这个以右值引用参数的 AcceptRvalueRef 函数中获得最大的收益。事实上，右值引用的来由从来就跟移动语义紧紧相关。这是右值存在的一个最大的价值（另外一个价值是用于转发，我们会在后面的小节中看到）。

对于本例而言，很有趣的是，读者也可以思考一下：如果我们不声明移动构造函数，而只声明一个常量左值的构造函数会发生什么？如同我们刚才提到的，常量左值引用是个“万

能”的引用类型，无论左值还是右值，常量还是非常量，一概能够绑定。那么如果 Copyable 没有移动构造函数，下列语句：

```
Copyable news = std::move(s);
```

将调用以常量左值引用为参数的拷贝构造函数。这是一种非常安全的设计——移动不成，至少还可以执行拷贝。因此，通常情况下，程序员会为声明了移动构造函数的类声明一个常量左值为参数的拷贝构造函数，以保证在移动构造不成时，可以使用拷贝构造（不过，我们也会在之后看到一些特殊用途的反例）。

为了语义的完整，C++11 中还存在着常量右值引用，比如我们通过以下代码声明一个常量右值引用。

```
const T && crvalueref = ReturnRvalue();
```

但是，一来右值引用主要就是为了移动语义，而移动语义需要右值是可以被修改的，那么常量右值引用在移动语义中就没有用武之处；二来如果要引用右值且让右值不可以更改，常量左值引用往往就足够了。因此在现在的情况下，我们还没有看到常量右值引用有何用处。

表 3-1 中，我们列出了在 C++11 中各种引用类型可以引用的值的类型。值得注意的是，只要能够绑定右值的引用类型，都能够延长右值的生命期。

表 3-1　C++11 中引用类型及其可以引用的值类型

引用类型	可以引用的值类型				注　记
	非常量左值	常量左值	非常量右值	常量右值	
非常量左值引用	Y	N	N	N	无
常量左值引用	Y	Y	Y	Y	全能类型，可用于拷贝语义
非常量右值引用	N	N	Y	N	用于移动语义、完美转发
常量右值引用	N	N	Y	Y	暂无用途

有的时候，我们可能不知道一个类型是否是引用类型，以及是左值引用还是右值引用（这在模板中比较常见）。标准库在 <type_traits> 头文件中提供了 3 个模板类：is_rvalue_reference、is_lvalue_reference、is_reference，可供我们进行判断。比如：

```
cout << is_rvalue_reference<string &&>::value;
```

我们通过模板类的成员 value 就可以打印出 stirng && 是否是一个右值引用了。配合第 4 章中的类型推导操作符 decltype，我们甚至还可以对变量的类型进行判断。当读者搞不清楚引用类型的时候，不妨使用这样的小工具实验一下。

3.3.4 std::move：强制转化为右值

在C++11中，标准库在<utility>中提供了一个有用的函数std::move，这个函数的名字具有迷惑性，因为实际上std::move并不能移动任何东西，它唯一的功能是将一个左值强制转化为右值引用，继而我们可以通过右值引用使用该值，以用于移动语义。从实现上讲，std::move基本等同于一个类型转换：

```
static_cast<T&&>(lvalue);
```

值得一提的是，被转化的左值，其生命期并没有随着左右值的转化而改变。如果读者期望std::move转化的左值变量lvalue能立即被析构，那么肯定会失望了。我们来看代码清单3-21所示的例子。

代码清单3-21

```
#include <iostream>
using namespace std;

class Moveable{
public:
    Moveable():i(new int(3)) {}
    ~Moveable() { delete i; }
    Moveable(const Moveable & m): i(new int(*m.i)) { }
    Moveable(Moveable && m):i(m.i) {
        m.i = nullptr;
    }
    int* i;
};

int main() {
    Moveable a;

    Moveable c(move(a));     // 会调用移动构造函数
    cout << *a.i << endl;    // 运行时错误
}
// 编译选项 :g++ -std=c++11 3-3-6.cpp -fno-elide-constructors
```

在代码清单3-21中，我们为类型Moveable定义了移动构造函数。这个函数定义本身没有什么问题，但调用的时候，使用了Moveable c(move(a));这样的语句。这里的a本来是一个左值变量，通过std::move将其转换为右值。这样一来，a.i就被c的移动构造函数设置为指针空值。由于a的生命期实际要到main函数结束才结束，那么随后对表达式*a.i进行计算的时候，就会发生严重的运行时错误。

这是个典型误用std::move的例子。当然，标准库提供该函数的目的不是为了让程序员搬起石头砸自己的脚。事实上，要使用该函数，必须是程序员清楚需要转换的时候。比如上例中，程序员应该知道被转化为右值的a不可以再使用。不过更多地，我们需要转换成为右

值引用的还是一个确实生命期即将结束的对象。我们来看看代码清单 3-22 所示的正确例子。

代码清单 3-22

```
#include <iostream>
using namespace std;

class HugeMem{
public:
    HugeMem(int size): sz(size > 0 ? size : 1) {
        c = new int[sz];
    }
    ~HugeMem() { delete [] c; }
    HugeMem(HugeMem && hm): sz(hm.sz), c(hm.c) {
        hm.c = nullptr;
    }
    int * c;
    int sz;
};
class Moveable{
public:
    Moveable():i(new int(3)), h(1024) {}
    ~Moveable() { delete i; }
    Moveable(Moveable && m):
        i(m.i), h(move(m.h)) {        // 强制转为右值，以调用移动构造函数
        m.i = nullptr;
    }
    int* i;
    HugeMem h;
};

Moveable GetTemp() {
    Moveable tmp = Moveable();
    cout << hex << "Huge Mem from " << __func__
        << " @" << tmp.h.c << endl; // Huge Mem from GetTemp @0x603030
    return tmp;
}

int main() {
    Moveable a(GetTemp());
    cout << hex << "Huge Mem from " << __func__
        << " @" << a.h.c << endl;   // Huge Mem from main @0x603030
}
// 编译选项 :g++ -std=c++11 3-3-7.cpp -fno-elide-constructors
```

在代码清单 3-22 中，我们定义了两个类型：HugeMem 和 Moveable，其中 Moveable 包含了一个 HugeMem 的对象。在 Moveable 的移动构造函数中，我们就看到了 std::move 函数的使用。该函数将 m.h 强制转化为右值，以迫使 Moveable 中的 h 能够实现移动构造。这里

可以使用 std::move，是因为 m.h 是 m 的成员，既然 m 将在表达式结束后被析构，其成员也自然会被析构，因此不存在代码清单 3-21 中的生存期不对的问题。另外一个问题可能是 std::move 使用的必要性。这里如果不使用 std::move(m.h) 这样的表达式，而是直接使用 m.h 这个表达式将会怎样？

其实这是 C++11 中有趣的地方：可以接受右值的右值引用本身却是个左值。这里的 m.h 引用了一个确定的对象，而且 m.h 也有名字，可以使用 &m.h 取到地址，因此是个不折不扣的左值。不过这个左值确确实实会很快“灰飞烟灭”，因为拷贝构造函数在 Moveable 对象 a 的构造完成后也就结束了。那么这里使用 std::move 强制其为右值就不会有问题了。而且，如果我们不这么做，由于 m.h 是个左值，就会导致调用 HugeMem 的拷贝构造函数来构造 Moveable 的成员 h（虽然这里没有声明，读者可以自行添加实验一下）。如果是这样，移动语义就没有能够成功地向类的成员传递。换言之，还是会由于拷贝而导致一定的性能上的损失。

事实上，为了保证移动语义的传递，程序员在编写移动构造函数的时候，应该总是记得使用 std::move 转换拥有形如堆内存、文件句柄等资源的成员为右值，这样一来，如果成员支持移动构造的话，就可以实现其移动语义。而即使成员没有移动构造函数，那么接受常量左值的构造函数版本也会轻松地实现拷贝构造，因此也不会引起大的问题。

3.3.5 移动语义的一些其他问题

我们在前面多次提到，移动语义一定是要修改临时变量的值。那么，如果这样声明移动构造函数：

```
Moveable(const Moveable &&)
```

或者这样声明函数：

```
const Moveable ReturnVal();
```

都会使得的临时变量常量化，成为一个常量右值，那么临时变量的引用也就无法修改，从而导致无法实现移动语义。因此程序员在实现移动语义一定要注意排除不必要的 const 关键字。

在 C++11 中，拷贝 / 移动构造函数实际上有以下 3 个版本：

```
T Object(T &)
T Object(const T &)
T Object(T &&)
```

其中常量左值引用的版本是一个拷贝构造版本，而右值引用版本是一个移动构造版本。默认情况下，编译器会为程序员隐式地生成一个（隐式表示如果不被使用则不生成）移动构造函数。不过如果程序员声明了自定义的拷贝构造函数、拷贝赋值函数、移动赋值函数、析

构函数中的一个或者多个，编译器都不会再为程序员生成默认版本。默认的移动构造函数实际上跟默认的拷贝构造函数一样，只能做一些按位拷贝的工作。这对实现移动语义来说是不够的。通常情况下，如果需要移动语义，程序员必须自定义移动构造函数。当然，对一些简单的、不包含任何资源的类型来说，实现移动语义与否都无关紧要，因为对这样的类型而言，移动就是拷贝，拷贝就是移动。

同样地，声明了移动构造函数、移动赋值函数、拷贝赋值函数和析构函数中的一个或者多个，编译器也不会再为程序员生成默认的拷贝构造函数。所以在 C++11 中，拷贝构造 / 赋值和移动构造 / 赋值函数必须同时提供，或者同时不提供，程序员才能保证类同时具有拷贝和移动语义。只声明其中一种的话，类都仅能实现一种语义。

其实，只实现一种语义在类的编写中也是非常常见的。比如说只有拷贝语义的类型——事实上在 C++11 之前我们见过大多数的类型的构造都是只使用拷贝语义的。而只有移动语义的类型则非常有趣，因为只有移动语义表明该类型的变量所拥有的资源只能被移动，而不能被拷贝。那么这样的资源必须是唯一的。因此，只有移动语义构造的类型往往都是“资源型”的类型，比如说智能指针，文件流等，都可以视为“资源型”的类型。在本书的第 5 章中，就可以看到标准库中的仅可移动的模板类：unique_ptr。一些编译器，如 vs2011，现在也把 ifstream 这样的类型实现为仅可移动的。

在标准库的头文件 <type_traits> 里，我们还可以通过一些辅助的模板类来判断一个类型是否是可以移动的。比如 is_move_constructible、is_trivially_move_constructible、is_nothrow_move_constructible，使用方法仍然是使用其成员 value。比如：

```
cout << is_move_constructible<UnknownType>::value;
```

就可以打印出 UnknowType 是否可以移动，这在一些情况下还是非常有用的。

而有了移动语义，还有一个比较典型的应用是可以实现高性能的置换（swap）函数。看看下面这段 swap 模板函数代码：

```
template <class T>
void swap(T& a, T& b)
{
    T tmp(move(a));
    a = move(b);
    b = move(tmp);
}
```

如果 T 是可以移动的，那么移动构造和移动赋值将会被用于这个置换。代码中，a 先将自己的资源交给 tmp，随后 b 再将资源交给 a，tmp 随后又将从 a 中得到的资源交给 b，从而完成了一个置换动作。整个过程，代码都只会按照移动语义进行指针交换，不会有资源的释放与申请。而如果 T 不可移动却是可拷贝的，那么拷贝语义会被用来进行置换。这就跟普通的置换语句是相同的了。因此在移动语义的支持下，我们仅仅通过一个通用的模板，就可能

更高效地完成置换，这对于泛型编程来说，无疑是具有积极意义的。

另外一个关于移动构造的话题是异常。对于移动构造函数来说，抛出异常有时是件危险的事情。因为可能移动语义还没完成，一个异常却抛出来了，这就会导致一些指针就成为悬挂指针。因此程序员应该尽量编写不抛出异常的移动构造函数，通过为其添加一个 noexcept 关键字，可以保证移动构造函数中抛出来的异常会直接调用 terminate 程序终止运行，而不是造成指针悬挂的状态。而标准库中，我们还可以用一个 std::move_if_noexcept 的模板函数替代 move 函数。该函数在类的移动构造函数没有 noexcept 关键字修饰时返回一个左值引用从而使变量可以使用拷贝语义，而在类的移动构造函数有 noexcept 关键字时，返回一个右值引用，从而使变量可以使用移动语义。我们来看一下代码清单 3-23 所示的例子。

代码清单 3-23

```
#include <iostream>
#include <utility>
using namespace std;

struct Maythrow {
    Maythrow() {}
    Maythrow(const Maythrow&) {
        std::cout << "Maythorow copy constructor." << endl;
    }
    Maythrow(Maythrow&&) {
        std::cout << "Maythorow move constructor." << endl;
    }
};

struct Nothrow {
    Nothrow() {}
    Nothrow(Nothrow&&) noexcept {
        std::cout << "Nothorow move constructor." << endl;
    }
    Nothrow(const Nothrow&) {
        std::cout << "Nothorow move constructor." << endl;
    }
};

int main() {
    Maythrow m;
    Nothrow n;

    Maythrow mt = move_if_noexcept(m);  // Maythorow copy constructor.
    Nothrow nt = move_if_noexcept(n);   // Nothorow move constructor.
    return 0;
}
// 编译选项 :g++ -std=c++11 3-3-8.cpp
```

在代码清单 3-23 中，可以清楚地看到 move_if_noexcept 的效果。事实上，move_if_noexcept 是以牺牲性能保证安全的一种做法，而且要求类的开发者对移动构造函数使用 noexcept 进行描述，否则就会损失更多的性能。这是库的开发者和使用者必须协同平衡考虑的。

还有一个与移动语义看似无关，但偏偏有些关联的话题是，编译器中被称为 RVO/NRVO 的优化（RVO, Return Value Optimization，返回值优化，或者 NRVO，Named Return Value optimization)。事实上，在本节中大量的代码都使用了 -fno-elide-constructors 选项在 g++/clang++ 中关闭这个优化，这样可以使读者在代码中较为容易地利用函数返回的临时量右值。

但若在编译的时候不使用该选项的话，读者会发现很多构造和移动都被省略了。对于下面这样的代码，一旦打开 g++/clang++ 的 RVO/NRVO，从 ReturnValue 函数中 a 变量拷贝 / 移动构造临时变量，以及从临时变量拷贝 / 移动构造 b 的二重奏就通通没有了。

```
A ReturnRvalue() { A a(); return a; }
A b = ReturnRvalue();
```

b 变量实际就使用了 ReturnRvalue 函数中 a 的地址，任何的拷贝和移动都没有了。通俗地说，就是 b 变量直接“霸占”了 a 变量。这是编译器中一个效果非常好的一个优化。不过 RVO/NRVO 并不是对任何情况都有效。比如有些情况下，一些构造是无法省略的。还有一些情况，即使 RVO/NRVO 完成了，也不能达到最好的效果。但结论是明显的，移动语义可以解决编译器无法解决的优化问题，因而总是有用的。

3.3.6 完美转发

所谓完美转发（perfect forwarding），是指在函数模板中，完全依照模板的参数的类型，将参数传递给函数模板中调用的另外一个函数。比如：

```
template <typename T>
void IamForwording(T t) {    IrunCodeActually(t);    }
```

这个简单的例子中，IamForwording 是一个转发函数模板。而函数 IrunCodeActually 则是真正执行代码的目标函数。对于目标函数 IrunCodeActually 而言，它总是希望转发函数将参数按照传入 Iamforwarding 时的类型传递（即传入 IamForwording 的是左值对象，IrunCodeActually 就能获得左值对象，传入 IamForwording 的是右值对象，IrunCodeActually 就能获得右值对象），而不产生额外的开销，就好像转发者不存在一样。

这似乎是一件非常容易的事情，但实际却并不简单。在上面例子中，我在 IamForwording 的参数中使用了最基本类型进行转发，该方法会导致参数在传给 IrunCodeActually 之前就产生了一次额外的临时对象拷贝。因此这样的转发只能说是正确的转发，但谈不上完美。

所以通常程序员需要的是一个引用类型，引用类型不会有拷贝的开销。其次，则需要考虑转发函数对类型的接受能力。因为目标函数可能需要能够既接受左值引用，又接受右值引

用。那么如果转发函数只能接受其中的一部分，我们也无法做到完美转发。结合表 3-1，我们会想到“万能”的常量左值类型。不过以常量左值为参数的转发函数却会遇到一些尴尬，比如：

```
void IrunCodeActually(int t){}
template <typename T>
void IamForwording(const T & t) {    IrunCodeActually(t);     }
```

这里，由于目标函数的参数类型是非常量左值引用类型，因此无法接受常量左值引用作为参数，这样一来，虽然转发函数的接受能力很高，但在目标函数的接受上却出了问题。那么我们可能就需要通过一些常量和非常量的重载来解决目标函数的接受问题。这在函数参数比较多的情况下，就会造成代码的冗余。而且依据表 3-1，如果我们的目标函数的参数是个右值引用的话，同样无法接受任何左值类型作为参数，间接地，也就导致无法使用移动语义。

那 C++11 是如何解决完美转发的问题的呢？实际上，C++11 是通过引入一条所谓“引用折叠”（reference collapsing）的新语言规则，并结合新的模板推导规则来完成完美转发。

在 C++11 以前，形如下列语句：

```
typedef const int T;
typedef T& TR;
TR& v = 1;     // 该声明在 C++98 中会导致编译错误
```

其中 TR& v = 1 这样的表达式会被编译器认为是不合法的表达式，而在 C++11 中，一旦出现了这样的表达式，就会发生引用折叠，即将复杂的未知表达式折叠为已知的简单表达式，具体如表 3-2 所示。

表 3-2　C++11 中的引用折叠规则

TR 的类型定义	声明 v 的类型	v 的实际类型
T&	TR	A&
T&	TR&	A&
T&	TR&&	A&
T&&	TR	A&&
T&&	TR&	A&
T&&	TR&&	A&&

这个规则并不难记忆，因为一旦定义中出现了左值引用，引用折叠总是优先将其折叠为左值引用。而模板对类型的推导规则就比较简单，当转发函数的实参是类型 X 的一个左值引用，那么模板参数被推导为 X& 类型，而转发函数的实参是类型 X 的一个右值引用的话，那么模板的参数被推导为 X&& 类型。结合以上的引用折叠规则，就能确定出参数的实际类型。进一步，我们可以把转发函数写成如下形式：

```
template <typename T>
```

```
void IamForwording(T && t) {
   IrunCodeActually(static_cast<T &&>(t));
}
```

注意 对于完美转发而言，右值引用并非“天生神力”，只是C++11新引入了右值，因此为其新定下了引用折叠的规则，以满足完美转发的需求。

注意一下，我们不仅在参数部分使用了T &&这样的标识，在目标函数传参的强制类型转换中也使用了这样的形式。比如我们调用转发函数时传入了一个X类型的左值引用，可以想象，转发函数将被实例化为如下形式：

```
void IamForwording(X& && t) {
   IrunCodeActually(static_cast<X& &&>(t));
}
```

应用上引用折叠规则，就是：

```
void IamForwording(X& t) {
   IrunCodeActually(static_cast<X&>(t));
}
```

这样一来，我们的左值传递就毫无问题了。实际使用的时候，IrunCodeActually如果接受左值引用的话，就可以直接调用转发函数。不过读者可能发现，这里调用前的static_cast没有什么作用。事实上，这里的static_cast是留给传递右值用的。

而如果我们调用转发函数时传入了一个X类型的右值引用的话，我们的转发函数将被实例化为：

```
void IamForwording(X&& && t) {
   IrunCodeActually(static_cast<X&& &&>(t));
}
```

应用上引用折叠规则，就是：

```
void IamForwording(X&& t) {
   IrunCodeActually(static_cast<X&&>(t));
}
```

这里我们就看到了static_cast的重要性。如我们在上面几个小节中讲到的，对于一个右值而言，当它使用右值引用表达式引用的时候，该右值引用却是个不折不扣的左值，那么我们想在函数调用中继续传递右值，就需要使用std::move来进行左右值的转换。而std::move通常就是一个static_cast。不过在C++11中，用于完美转发的函数却不再叫作move，而是另外一个名字：forward。所以我们可以把转发函数写成这样：

```
template <typename T>
void IamForwording(T && t) {
   IrunCodeActually(forward(t));
```

```
}
```

move 和 forward 在实际实现上差别并不大。不过标准库这么设计，也许是为了让每个名字对应于不同的用途，以应对未来可能的扩展（虽然现在我们使用 move 可能也能通过完美转发函数的编译，但这并不是推荐的做法）。

我们来看一个完美转发的例子，如代码清单 3-24 所示。

代码清单 3-24

```
#include <iostream>
using namespace std;

void RunCode(int && m) { cout << "rvalue ref" << endl; }
void RunCode(int & m) { cout << "lvalue ref" << endl; }
void RunCode(const int && m) { cout << "const rvalue ref" << endl; }
void RunCode(const int & m) { cout << "const lvalue ref" << endl; }

template <typename T>
void PerfectForward(T &&t) {   RunCode(forward<T>(t));    }

int main() {
    int a;
    int b;
    const int c = 1;
    const int d = 0;
    PerfectForward(a);            // lvalue ref
    PerfectForward(move(b));      // rvalue ref
    PerfectForward(c);            // const lvalue ref
    PerfectForward(move(d));      // const rvalue ref
}
// 编译选项 :g++ -std=c++11 3-3-9.cpp
```

在代码清单 3-24 中，我们使用了表 3-1 中的所有 4 种类型的值对完美转发进行测试，可以看到，所有的转发都被正确地送到了目的地。

完美转发的一个作用就是做包装函数，这是一个很方便的功能。我们对代码清单 3-24 中的转发函数稍作修改，就可以用很少的代码记录单参数函数的参数传递状况，如代码清单 3-25 所示。

代码清单 3-25

```
#include <iostream>
using namespace std;

template <typename T, typename U>
void PerfectForward(T &&t, U& Func) {
```

```
    cout << t << "\tforwarded..." << endl;
    Func(forward<T>(t));
}

void RunCode(double && m) {}
void RunHome(double && h) {}
void RunComp(double && c) {}

int main() {
    PerfectForward(1.5, RunComp);    // 1.5      forwarded...
    PerfectForward(8, RunCode);      // 8        forwarded...
    PerfectForward(1.5, RunHome);    // 1.5      forwarded...
}
// 编译选项：g++ -std=c++11 3-3-10.cpp
```

当然，读者可以尝试将该例子变得更复杂一点，以更加符合实际的需求。事实上，在C++11标准库中我们可以看到大量完美转发的实际应用，一些很小巧好用的函数，比如make_pair、make_unique等在C++11都通过完美转发实现了。这样一来，就减少了一些函数版本的重复（const和非const版本的重复），并能够充分利用移动语义。无论从运行性能的提高还是从代码编写的简化上，完美转发都堪称完美。

3.4 显式转换操作符

☞ 类别：库作者

在C++中，有个非常好也非常坏的特性，就是隐式类型转换。隐式类型转换的“自动性”可以让程序员免于层层构造类型。但也是由于它的自动性，会在一些程序员意想不到的地方出现严重的但不易被发现的错误。我们可以先看看代码清单3-26所示的这个例子。

代码清单3-26

```
#include <iostream>
using namespace std;

struct Rational1 {
    Rational1(int n = 0, int d = 1): num(n), den(d) {
        cout << __func__ << "(" << num << "/" << den << ")" << endl;
    }
    int num;     // Numerator（被除数）
    int den;     // Denominator（除数）
};

struct Rational2 {
    explicit Rational2(int n = 0, int d = 1): num(n), den(d) {
        cout << __func__ << "(" << num << "/" << den << ")" << endl;
    }
```

```
    int num;
    int den;
};

void Display1(Rational1 ra){
    cout << "Numerator: " << ra.num <<" Denominator: "<< ra.den <<endl;
}
void Display2(Rational2 ra){
    cout << "Numerator: "<< ra.num <<" Denominator: "<< ra.den<<endl;
}

int main(){
    Rational1 r1_1 = 11;                    // Rational1(11/1)
    Rational1 r1_2(12);                     // Rational1(12/1)

    Rational2 r2_1 = 21;                    // 无法通过编译
    Rational2 r2_2(22);                     // Rational2(22/1)

    Display1(1);                            // Rational1(1/1)
                                            // Numerator: 1 Denominator: 1
    Display2(2);                            // 无法通过编译
    Display2(Rational2(2));                 // Rational2(2/1)
                                            // Numerator: 2 Denominator: 1
    return 0;
}
// 编译选项 :g++ -std=c++11 3-4-1.cpp
```

在代码清单 3-26 中，声明了两个类型 Rational1 和 Rational2。两者在代码上的区别不大，只不过 Rational1 的构造函数 Rational1(int,int) 没有 explicit 关键字修饰，这意味着该构造函数可以被隐式调用。因此，在定义变量 r1_1 的时候，字面量 11 就会成功地构造出 Rational1(11, 1) 这样的变量，Rational2 却不能从字面量 21 中构造，这是因为其构造函数由于使用了关键字 explicit 修饰，禁止被隐式构造，因此会导致编译失败。相同的情况也出现在函数 Display2 上，由于字面量 2 不能隐式地构造出 Rational2 对象，因此表达式 Display2(2) 的编译同样无法通过。

这里虽然 Display1(1) 编译成功，不过如果不是结合了上面 Rational1 的定义，我们很容易在阅读代码的时候产生误解。按照习惯，程序员会误认为 Display1 是个打印整型数的函数。因此，使用了 explicit 这个关键字保证对象的显式构造在一些情况下都是必须的。

不过同样的机制并没有出现在自定义的类型转换符上。这就允许了一个逆向的过程，从自定义类型转向一个已知类型。这样虽然出现问题的几率远小于从已知类型构造自定义类型，不过有的时候，我们确实应该阻止会产生歧义的隐式转换。让我们来看看代码清单 3-27 所示的例子，该例子来源于 C++11 提案。

代码清单 3-27

```
#include <iostream>
using namespace std;

template <typename T>
class Ptr {
public:
    Ptr(T* p): _p(p) {}
    operator bool() const {
        if (_p != 0)
            return true;
        else
            return false;
    }
private:
    T* _p;
};

int main() {
    int a;
    Ptr<int> p(&a);

    if (p)       // 自动转换为 bool 型，没有问题
        cout << "valid pointer." << endl;    // valid pointer.
    else
        cout << "invalid pointer." << endl;

    Ptr<double> pd(0);
    cout << p + pd << endl; // 1，相加，语义上没有意义
}
// 编译选项:g++ 3-4-2.cpp
```

在代码清单 3-27 中，我们定义了一个指针模板类型 Ptr。为了方便判断指针是否有效，我们为指针编写了自定义类型转换到 bool 类型的函数，这样一来，我们就可以通过 if(p) 这样的表达式来轻松地判断指针是否有效。不过这样的转换使得 Ptr<int> 和 Ptr<double> 两个指针的加法运算获得了语法上的允许。不过明显地，我们无法看出其语义上的意义。

在 C++11 中，标准将 explicit 的使用范围扩展到了自定义的类型转换操作符上，以支持所谓的“显式类型转换”。explicit 关键字作用于类型转换操作符上，意味着只有在直接构造目标类型或显式类型转换的时候可以使用该类型。我们可以看看代码清单 3-28 所示的例子。

代码清单 3-28

```
class ConvertTo {};
class Convertable {
public:
    explicit operator ConvertTo () const { return ConvertTo(); }
```

```
};
void Func(ConvertTo ct) {}
void test() {
    Convertable c;
    ConvertTo ct(c);          // 直接初始化，通过
    ConvertTo ct2 = c;        // 拷贝构造初始化，编译失败
    ConvertTo ct3 = static_cast<ConvertTo>(c);  // 强制转化，通过
    Func(c);                  // 拷贝构造初始化，编译失败
}
// 编译选项：g++ -std=c++11 3-4-3.cpp
```

在代码清单 3-28 中，我们定义了两个类型 ConvertTo 和 Convertable，Convertable 定义了一个显式转换到 ConvertTo 类型的类型转换符。那么对于 main 中 ConvertTo 类型的 ct 变量而言，由于其直接初始化构造于 Convertable 变量 c，所以可以编译通过。而做强制类型转换的 ct3 同样通过了编译。而 ct2 由于需要从 c 中拷贝构造，因而不能通过编译。此外，我们使用函数 Func 的时候，传入 Convertable 的变量 c 的也会导致参数的拷贝构造，因此也不能通过编译。

如果我们把该方法用于代码清单 3-27 中，可以发现我们预期的事情就发生了，if(p) 可以通过编译，因为可以通过 p 直接构造出 bool 类型的变量。而 p + pd 这样的语句就无法通过编译了，这是由于全局的 operator + 并不接受 bool 类型变量为参数，而 Convertable 也不能直接构造出适用于 operator + 的 int 类型的变量造成的（不过读者可以尝试一下使用 p && pd 这样的表达式，是能够通过编译的）。这样一来，程序的行为将更加良好。

可以看到，所谓显式类型转换并没完全禁止从源类型到目标类型的转换，不过由于此时拷贝构造和非显式类型转换不被允许，那么我们通常就不能通过赋值表达式或者函数参数的方式来产生这样一个目标类型。通常通过赋值表达式和函数参数进行的转换有可能是程序员的一时疏忽，而并非本意。那么使用了显式类型转换，这样的问题就会暴露出来，这也是我们需要显式转换符的一个重要原因。

3.5 列表初始化

☞ **类别：所有人**

3.5.1 初始化列表

在 C++98 中，标准允许使用花括号 "{}" 对数组元素进行统一的集合（列表）初始值设定，比如：

```
int arr[5] = {0};
int arr[] = {1, 2, 3, 4} ;
```

这些都是合法的表达式。不过一些自定义类型，却无法享受这样便利的初始化。通常，

如标准程序库中的 vector 这样的容器，总是需要声明对象 - 循环初始化这样的重复动作，这对于使用模板的泛型编程无疑是非常不利的。

在 2.7 节中，我们看到了 C++11 对类成员的快速就地初始化。有一种初始化形式就是使用花括号的集合（列表）初始化。而事实上，在 C++11 中，集合（列表）的初始化已经成为 C++ 语言的一个基本功能，在 C++11 中，这种初始化的方法被称为“初始化列表”（initializer list）。让我们来看看代码清单 3-29 所示的这个例子。

代码清单 3-29

```
#include <vector>
#include <map>
using namespace std;

int a[] = {1, 3, 5};          // C++98 通过，C++11 通过
int b[] {2, 4, 6};            // C++98 失败，C++11 通过
vector<int> c{1, 3, 5};       // C++98 失败，C++11 通过
map<int, float> d =
     {{1, 1.0f}, {2, 2.0f} , {5, 3.2f}}; // C++98 失败，C++11 通过
 // 编译选项 :g++ -c -std=c++11 3-5-1.cpp
```

在代码清单 3-29 中，我们看到了变量 b、c、d，在 C++98 的情况下均无法通过编译，在 C++11 中，却由于列表初始化的存在而可以通过编译。这里，列表初始化可以在“{}”花括号之前使用等号，其效果与不带使用等号的初始化相同。

这样一来，自动变量和全局变量的初始化在 C++11 中被丰富了。程序员可以使用以下几种形式完成初始化的工作：

- 等号“=”加上赋值表达式（assignment-expression），比如 int a = 3 + 4。
- 等号“=”加上花括号式的初始化列表，比如 int a = {3 + 4}。
- 圆括号式的表达式列表（expression-list），比如 int a (3 + 4)。
- 花括号式的初始化列表，比如 int a {3 + 4}。

而后两种形式也可以用于获取堆内存 new 操作符中，比如：

```
int * i = new int(1);
double * d = new double{1.2f};
```

这在 C++11 中也是合法的表达式。

代码清单 3-29 中可能令读者比较惊讶的是，使用初始化列表对 vector、map 等非内置的复杂的数据类型进行初始化竟然也是可以的。进一步地，读者可能会猜测是否初始化列表是专属于内置类型、数组，以及标准模板库中容器的功能呢？

事实并非如此，如同我们所提到的，在C++11中，标准总是倾向于使用更为通用的方式来支持新的特性。标准模板库中容器对初始化列表的支持源自<initializer_list>这个头文件中initialize_list类模板的支持。程序员只要#include了<initializer_list>头文件，并且声明一个以initialize_list<T>模板类为参数的构造函数，同样可以使得自定义的类使用列表初始化。让我们来看一看代码清单3-30的例子。

代码清单3-30

```
#include <vector>
#include <string>
using namespace std;

enum Gender {boy, girl};
class People {
public:
    People(initializer_list<pair<string, Gender>> l) { // initializer_list的构造函数
        auto i = l.begin();
        for (;i != l.end(); ++i)
            data.push_back(*i);
    }

private:
    vector<pair<string, Gender>> data;
};

People ship2012 = {{"Garfield", boy}, {"HelloKitty", girl}};
// 编译选项:g++ -c -std=c++11 3-5-2.cpp
```

在代码清单3-30中，我们为类People定义了一个使用initializer_list<pair<string, Gender>>模板类作为参数的构造函数。这里我们使用了C++11的auto关键字来自动类型推导以简化代码的编写（其意义比较明显，这里就不展开解释了，详情请查看4.2节）。由于该构造函数的存在，ship2012声明就可以使用列表初始化了。事实上，编写一个列表初始化的构造函数并不困难。对于旧有的代码，列表初始化构造函数还常常可以调用已有的代码来实现。

同样的，函数的参数列表也可以使用初始化列表，如代码清单3-31所示。

代码清单3-31

```
#include <initializer_list>
using namespace std;

void Fun(initializer_list<int> iv){ }

int main() {
    Fun({1, 2});
    Fun({}); // 空列表
}
```

```
// 编译选项 :g++ -std=c++11 3-5-3.cpp
```

在代码清单 3-31 中，定义了一个可以接受初始化列表的函数 Fun。同理，类和结构体的成员函数也可以使用初始化列表，包括一些操作符的重载函数。而在代码清单 3-32 所示的这个例子中，我们利用了初始化列表重载了 operator[]，并且重载了 operator = 以及使用辅助的数组。虽然这个例子比较复杂，但重载的效果还是能够让人感觉眼前一亮的。

代码清单 3-32

```
#include <iostream>
#include <vector>
using namespace std;

class Mydata {
public:
    Mydata & operator [] (initializer_list<int> l)
    {
        for (auto i = l.begin(); i != l.end(); ++i)
            idx.push_back(*i);
        return *this;
    }
    Mydata & operator = (int v)
    {
        if (idx.empty() != true) {
            for (auto i = idx.begin(); i != idx.end(); ++i) {
                d.resize((*i > d.size()) ? *i : d.size());
                d[*i - 1] = v;
            }
            idx.clear();
        }
        return *this;
    }

    void Print() {
        for (auto i = d.begin(); i != d.end(); ++i)
            cout << *i << " ";
        cout << endl;
    }

private:
    vector<int> idx;     // 辅助数组，用于记录 index
    vector<int> d;
};

int main() {
    Mydata d;
    d[{2, 3, 5}] = 7;
    d[{1, 4, 5, 8}] = 4;
```

```
    d.Print();  // 4 7 7 4 4 0 0 4
}

// 编译选项 :g++ -std=c++11 3-5-4.cpp
```

在代码清单 3-32 中，我们看到自定义类型 Mydata 拥有一个以前所有 C++ 代码都不具备的功能，即可以在 [] 符号中使用列表，将设置数组中的部分为一个指定的值。在这里我们先把数组的第 2、3、5 位设为数值 7，而后又将其 1、4、5、8 位设为数值 4，最终我们得到数组的内容为“4 7 7 4 4 0 0 4”。读者可以自行分析一下代码的实现方式（这段代码比较粗糙，读者应该重点体会初始化列表带来的编程上的灵活性）。当然，由于内置的数组不能重载 operator[]，我们也就无法为其实现相应的功能。

此外，初始化列表还可以用于函数返回的情况。返回一个初始化列表，通常会导致构造一个临时变量，比如：

```
vector<int> Func() { return {1, 3}; }
```

当然，跟声明时采用列表初始化一样，列表初始化构造成什么类型是依据返回类型的，比如：

```
deque<int> Func2() { return {3, 5};}
```

上面的返回值就是以 deque<int> 列表初始化构造函数而构造的。而跟普通的字面量相同，如果返回值是一个引用类型的话，则会返回一个临时变量的引用。比如：

```
const vector<int>& Func1() { return {3, 5};}
```

这里注意，必须要加 const 限制符。该规则与返回一个字面常量是一样的。

3.5.2 防止类型收窄

使用列表初始化还有一个最大优势是可以防止类型收窄（narrowing）。类型收窄一般是指一些可以使得数据变化或者精度丢失的隐式类型转换。可能导致类型收窄的典型情况如下：

- 从浮点数隐式地转化为整型数。比如：int a = 1.2，这里 a 实际保存的值为整数 1，可以视为类型收窄。
- 从高精度的浮点数转为低精度的浮点数，比如从 long double 隐式地转化为 double，或从 double 转为 float。如果这些转换导致精度降低，都可以视为类型收窄。
- 从整型（或者非强类型的枚举）转化为浮点型，如果整型数大到浮点数无法精确地表示，则也可以视为类型收窄。
- 从整型（或者非强类型的枚举）转化为较低长度的整型，比如：unsigned char = 1024，1024 明显不能被一般长度为 8 位的 unsigned char 所容纳，所以也可以视为类型收窄。

值得注意的是，如果变量 a 从类型 A 转化为类型 B，其值在 B 中也是可以被表示的，且再转化回类型 A 能获得原有的值的话，那么这种类型转换也不能叫作类型收窄。所以类型收窄也可以简单地理解为新类型无法表示原有类型数据的值的情况。事实上，发生类型收窄通常也是危险的，应引起程序员的注意。因此，在 C++11 中，使用初始化列表进行初始化的数据编译器是会检查其是否发生类型收窄的。我们来看看代码清单 3-33 所示的这个例子。

代码清单 3-33

```
const int x = 1024;
const int y = 10;

char a = x;                          // 收窄，但可以通过编译
char* b = new char(1024);            // 收窄，但可以通过编译

char c = {x};                        // 收窄，无法通过编译
char d = {y};                        // 可以通过编译
unsigned char e {-1};                // 收窄，无法通过编译

float f { 7 };                        // 可以通过编译
int g { 2.0f };                      // 收窄，无法通过编译
float * h = new float{1e48};          // 收窄，无法通过编译
float i = 1.2l;                       // 可以通过编译

// 编译选项 :clang++ -std=c++11 3-5-5.cpp
```

在例子代码清单 3-33 中，我们定义了 a 到 i 一共 9 个需要初始化的变量。可以看到，对于变量 a 和 *b 而言，由于其采用的是赋值表达符及圆括号式的表达式初始化，所以虽然它们的数据类型明显收窄（char 通常取值范围为 -128 到 127），却不会引发编译失败（事实上，在我们的实验机上会得到编译器的警告）。而使用初始化列表的情况则不一样。对于变量 c，由于其类型收窄，则会导致编译器报错。而对于变量 d 来说，其初始化使用了常量值 10，而 10 是可以由 char 类型表示的，因此这里不会发生收窄，编译可以通过。同样的情况还发生在变量 f、i 的初始化上。虽然初始化语句中的变量类型往往“大于”变量声明的类型，但是由于值在 f、i 中可以表示，还可以被转回原有类型不发生数据改变或者精度错误等，因此也不能算收窄。

比较容易引起疑问的是无符号类型的变量 e。虽然按理说 e 如果再被转换为有符号数，其值依然是 –1，但对于无符号数而言，并不能表示 –1，因此这里我们也认为 e 的初始化有收窄的情况。另外，f 和 g 的差别在于 2.0f 是一个有精度的浮点数值，通常可以认为，将 2.0f 转换成整型会丢失精度，所以 g 的声明也是收窄的。

在 C++11 中，列表初始化是唯一一种可以防止类型收窄的初始化方式。这也是列表初始化区别于其他初始化方式的地方。事实上，现有编译器大多数会在发生类型收窄的时候提示用户，因为类型收窄通常是代码可能出现问题的征兆。C++11 将列表初始化设定为可以防范

类型收窄，也就是为了加强类型使用的安全性。

总的来说，列表初始化改变了 C++ 中对类型初始化的一些基本模式，将标准程序库跟语言拉得更近了。这样的做法有效地统一了内置类型和自定义类型的行为。这也是 C++11 设计所遵循的一个思想，即通用为本，专用为末。

3.6 POD 类型

☞ **类别：部分人**

POD 是英文中 Plain Old Data 的缩写。POD 在 C++ 中是非常重要的一个概念，通常用于说明一个类型的属性，尤其是用户自定义类型的属性。POD 属性在 C++11 中往往又是构建其他 C++ 概念的基础，事实上，在 C++11 标准中，POD 出现的概率相当高。因此学习 C++，尤其是在 C++11 中，了解 POD 的概念是非常必要的。

POD 意如其名。Plain，表示了 POD 是个普通的类型，在 C++ 中常见的类型都有这样的属性，而不像一些存在着虚函数虚继承的类型那么特别。而 Old 则体现了其与 C 的兼容性，比如可以用最老的 memcpy() 函数进行复制，使用 memset() 进行初始化等。当然，这样的描述都太过于笼统，具体地，C++11 将 POD 划分为两个基本概念的合集，即：平凡的（trivial）和标准布局的（standard layout）。

我们先来看一下平凡的定义。通常情况下，一个平凡的类或结构体应该符合以下定义：

1）拥有平凡的默认构造函数（trivial constructor）和析构函数（trivial destructor）。

平凡的默认构造函数就是说构造函数“什么都不干”。通常情况下，不定义类的构造函数，编译器就会为我们生成一个平凡的默认构造函数。而一旦定义了构造函数，即使构造函数不包含参数，函数体里也没有任何的代码，那么该构造函数也不再是“平凡”的。比如：

```
struct NoTrivial { NoTrivial(); };
```

在 NoTrivial 的定义中，构造函数就不是平凡的，这对于析构函数来讲也类似。但这样的类型声明并非“无可救药”地“非平凡化”（non-trivial）了，在第 7 章中，可以看到如何使用 =default 关键字来显式地声明缺省版本的构造函数，从而使得类型恢复“平凡化”。

2）拥有平凡的拷贝构造函数（trivial copy constructor）和移动构造函数（trivial move constructor）。平凡的拷贝构造函数基本上等同于使用 memcpy 进行类型的构造。同平凡的默认构造函数一样，不声明拷贝构造函数的话，编译器会帮程序员自动地生成。同样地，可以显式地使用 =default 声明默认拷贝构造函数。

而平凡移动构造函数跟平凡的拷贝构造函数类似，只不过是用于移动语义。

3）拥有平凡的拷贝赋值运算符（trivial assignment operator）和移动赋值运算符（trivial

move operator）。这基本上与平凡的拷贝构造函数和平凡的移动构造运算符类似。

4）不能包含虚函数以及虚基类。

以上 4 点虽然看似复杂，不过在 C++11 中，我们可以通过一些辅助的类模板来帮我们进行以上属性的判断。

```
template <typename T> struct std::is_trivial;
```

类模板 is_trivial 的成员 value 可以用于判断 T 的类型是否是一个平凡的类型。除了类和结构体外，is_trivial 还可以对内置的标量类型数据（比如 int、float 都属于平凡类型）及数组类型（元素是平凡类型的数组总是平凡的）进行判断。

我们可以看看代码清单 3-34 所示的例子。

代码清单 3-34

```
#include <iostream>
#include <type_traits>
using namespace std;

struct Trivial1 {};
struct Trivial2 {
public:
    int a;
private:
    int b;
};

struct Trivial3 {
    Trivial1 a;
    Trivial2 b;
};

struct Trivial4 {
    Trivial2 a[23];
};

struct Trivial5 {
    int x;
    static int y;
};

struct NonTrivial1 {
    NonTrivial1() : z(42) {}
    int z;
};

struct NonTrivial2 {
    NonTrivial2();
```

```
    int w;
};
NonTrivial2::NonTrivial2() = default;

struct NonTrivial3 {
    Trivial5 c;
    virtual void f();
};

int main() {
    cout << is_trivial<Trivial1>::value << endl;     // 1
    cout << is_trivial<Trivial2>::value << endl;     // 1
    cout << is_trivial<Trivial3>::value << endl;     // 1
    cout << is_trivial<Trivial4>::value << endl;     // 1
    cout << is_trivial<Trivial5>::value << endl;     // 1
    cout << is_trivial<NonTrivial1>::value << endl; // 0
    cout << is_trivial<NonTrivial2>::value << endl; // 0
    cout << is_trivial<NonTrivial3>::value << endl; // 0
    return 0;
}

// 编译选项 :g++ -std=c++11 3-6-1.cpp
```

读者可以依照代码清单 3-34 的输出结果核对上面提到的 4 种规则。

POD 包含的另外一个概念是标准布局。标准布局的类或结构体应该符合以下定义：

1）所有非静态成员有相同的访问权限（public, private, protected）。

这一点非常好理解，比如：

```
struct {
public:
int a;
private:
    int b;
};
```

成员 a 和 b 就拥有不同的访问权限，因此该匿名结构体不是标准布局的。如果去掉 private 关键字的话，那么，该匿名结构体就符合标准布局的定义了。

```
struct {
public:
int a;
    int b;
};
```

2）在类或者结构体继承时，满足以下两种情况之一：

❑派生类中有非静态成员，且只有一个仅包含静态成员的基类。

❑基类有非静态成员，而派生类没有非静态成员。

这样的类或者结构体，也是标准布局的。比如下面的例子：

```
struct B1 { static int a; };
struct D1 : B1 { int d; };

struct B2 { int a; };
struct D2 : B2 { static int d; };

struct D3 : B2, B1 { static int d; };
struct D4 : B2 { int d; };
struct D5 : B2, D1 { };
```

D1、D2 和 D3 都是标准布局的，而 D4 和 D5 则不属于标准布局的。这实际上使得非静态成员只要同时出现在派生类和基类间，其即不属于标准布局的。而多重继承也会导致类型布局的一些变化，所以一旦非静态成员出现在多个基类中，派生类也不属于标准布局的。

3）类中第一个非静态成员的类型与其基类不同。

这条规则非常特别，用于形如：

```
struct A : B{ B b; };
```

这样的情况。这里 A 类型不是一个标准布局的类型，因为第一个非静态成员变量 b 的类型跟 A 所继承的类型 B 相同。而形如：

```
struct C : B {int a; B b;};
```

则是一个标准布局的类型。

读者可能对这个规则感到不解，不过该规则实际上是基于 C++ 中允许优化不包含成员的基类而产生的。我们可以看看代码清单 3-35 这个例子。

代码清单 3-35

```
#include <iostream>
using namespace std;

struct B1 {};
struct B2 {};

struct D1 : B1 {
    B1 b;        // 第一个非静态变量跟基类相同
    int i;
};

struct D2 : B1 {
    B2 b;
    int i;
```

```
};

int main() {
    D1 d1;
    D2 d2;
    cout << hex;
    cout << reinterpret_cast<long long>(&d1) << endl;
  // 7ffffd945c60
    cout << reinterpret_cast<long long>(&(d1.b)) << endl;
  // 7ffffd945c61
    cout << reinterpret_cast<long long>(&(d1.i)) << endl;
  // 7ffffd945c64

    cout << reinterpret_cast<long long>(&d2) << endl;
  // 7ffffd945c50
    cout << reinterpret_cast<long long>(&(d2.b)) << endl;
  // 7ffffd945c50
    cout << reinterpret_cast<long long>(&(d2.i)) << endl;
// 7ffffd945c54
}

// 编译选项 :g++ -std=c++11 3-6-2.cpp
```

在代码清单 3-35 中，我们声明了 4 个类。其中两个没有成员的基类 B1 和 B2，以及两个派生于 B1 的派生类 D1 和 D2。D1 和 D2 唯一的区别是第一个非静态成员的类型。在 D1 中，第一个非静态成员的类型是 B1，这跟它的基类相同；而 D2 中，第一个非静态成员的类型则是 B2。直观地看，D1 和 D2 应该是“布局相同”的，程序员应该可以使用 memcpy 这样的函数在这两种类型间进行拷贝，但实际上却并不是这样。

我们可以看看 main 函数中的状况。在 main 中，将 D1 类型的变量 d1 以及 D2 类型的变量 d2 的地址分别打印出来。同时我们也把它们的成员的地址打印出来。可以看到，对于 d2，它和它的成员共享了一个地址（本例中，实验机上的结果为 7ffffd945c50），而对于 d1 却没有出现类似的情况。

事实上，在 C++ 标准中，如果基类没有成员，标准允许派生类的第一个成员与基类共享地址。因为派生类的地址总是“堆叠”在基类之上的，所以这样的地址共享，表明了基类并没有占据任何的实际空间（可以节省一点数据）。但是如果基类的第一个成员仍然是基类，在我们的例子中可以看到，编译器仍然会为基类分配 1 字节的空间。分配为 1 字节空间是由于 C++ 标准要求类型相同的对象必须地址不同（基类地址及派生类中成员 d 的地址必须不同），而导致的结果是，对于 D1 和 D2 两种类型而言，其“布局”也就是不同的了。我们可以看看如图 3-3 所示的示意图。

所以在标准布局的解释中，C++11 标准强制要求派生类的第一个非静态成员的类型必须不同于基类。

4）没有虚函数和虚基类。

5）所有非静态数据成员均符合标准布局类型，其基类也符合标准布局。这是一个递归的定义，没有什么好特别解释的。

以上 5 点构成了标准布局的含义，最为重要的应该是前两条。

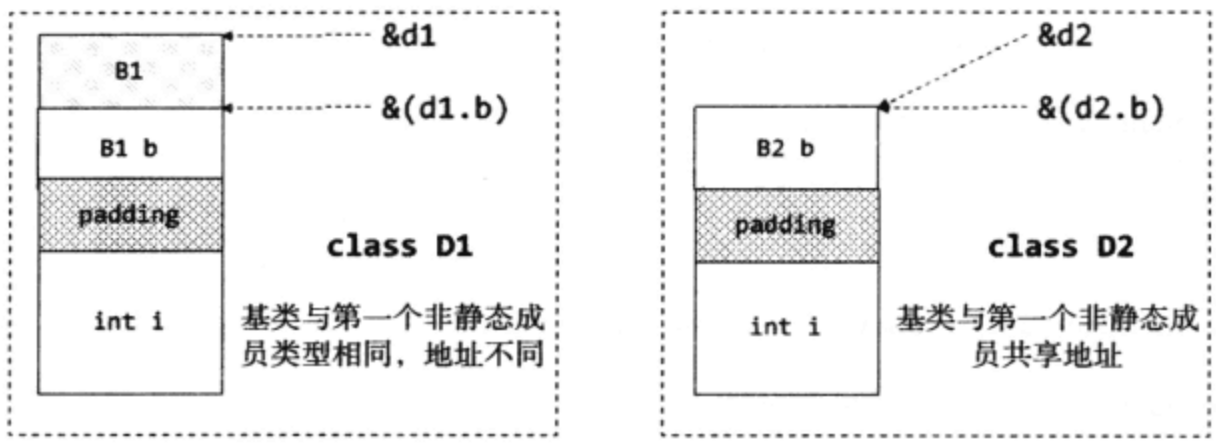


图 3-3　基类地址与派生类第一个非静态成员地址关系

同样，在 C++11 中，我们可以使用模板类来帮助判断类型是否是一个标准布局的类型。

```
template <typename T> struct std::is_standard_layout;
```

通过 is_standard_layout 模板类的成员 value（is_standard_layout<T>::value），我们可以在代码中打印出类型的标准布局属性。那么，通过代码清单 3-36 所示的这个例子，我们可以加深一下对标准类型的理解。

代码清单 3-36

```
#include <iostream>
#include <type_traits>
using namespace std;

struct SLayout1 {};

struct SLayout2 {
private:
    int x;
    int y;
};

struct SLayout3 : SLayout1 {
    int x;
    int y;
    void f();
};

struct SLayout4 : SLayout1 {
```

```
    int x;
    SLayout1 y;
};

struct SLayout5 : SLayout1, SLayout3 {};

struct SLayout6 { static int y; };

struct SLayout7: SLayout6 { int x; };

struct NonSLayout1 : SLayout1 {
    SLayout1 x;
    int i;
};

struct NonSLayout2 : SLayout2 { int z; };

struct NonSLayout3 : NonSLayout2 {};

struct NonSLayout4 {
public:
    int x;
private:
    int y;
};

int main() {
    cout<<is_standard_layout<SLayout1>::value << endl;  // 1
    cout<<is_standard_layout<SLayout2>::value << endl;  // 1
    cout<<is_standard_layout<SLayout3>::value << endl;  // 1
    cout<<is_standard_layout<SLayout4>::value << endl;  // 1
    cout<<is_standard_layout<SLayout5>::value << endl;  // 1
    cout<<is_standard_layout<SLayout6>::value << endl;  // 1
    cout<<is_standard_layout<SLayout7>::value << endl;  // 1

    cout<<is_standard_layout<NonSLayout1>::value << endl;   // 0
    cout<<is_standard_layout<NonSLayout2>::value << endl;   // 0
    cout<<is_standard_layout<NonSLayout3>::value << endl;   // 0
    cout<<is_standard_layout<NonSLayout4>::value << endl;   // 0
    return 0;
}

// 编译选项 :g++ -std=c++11 3-6-3.cpp
```

同样地，读者可以对照我们上述的 5 条规则，自行分析一下代码清单 3-36 中的情况。

那么，我们现在回到 POD 来，对于 POD 而言，在 C++11 中的定义就是平凡的和标准布局的两个方面。同样地，要判定某一类型是否是 POD，标准库中的 <type_traits> 头文件也为程序员提供了如下模板类：

```
template <typename T> struct std::is_pod;
```

我们可以使用 std::is_pod<T>::value 来判定一个类型是否是 POD，如代码清单 3-37 所示。

代码清单 3-37

```
#include <type_traits>
#include <iostream>
using namespace std;

union U{};
union U1{ U1(){} };
enum E{};
typedef double* DA;
typedef void (*PF)(int, double);

int main() {
    cout << is_pod<U>::value << endl;    // 1
    cout << is_pod<U1>::value << endl;   // 0
    cout << is_pod<E>::value << endl;    // 1
    cout << is_pod<int>::value << endl; // 1
    cout << is_pod<DA>::value << endl;   // 1
    cout << is_pod<PF>::value << endl;   // 1
}

// 编译选项 :g++ -std=c++11 3-6-4.cpp
```

事实上，如我们在代码清单 3-37 中看到的一样，很多内置类型默认都是 POD 的。POD 最为复杂的地方还是在类或者结构体的判断。不过通过上面平凡和标准布局的判断，相信读者对 POD 已经有所理解。那么，使用 POD 有什么好处呢？我们看得到的大概有如下 3 点：

1）字节赋值，代码中我们可以安全地使用 memset 和 memcpy 对 POD 类型进行初始化和拷贝等操作。

2）提供对 C 内存布局兼容。C++ 程序可以与 C 函数进行相互操作，因为 POD 类型的数据在 C 与 C++ 间的操作总是安全的。

3）保证了静态初始化的安全有效。静态初始化在很多时候能够提高程序的性能，而 POD 类型的对象初始化往往更加简单（比如放入目标文件的 .bss 段，在初始化中直接被赋 0）。

如我们所提到的，理解 POD 对理解 C++11 中其他概念非常重要，之后我们还会在本书中看到很多引用 POD 的地方。

3.7 非受限联合体

☞ **类别：部分人**

在 C/C++ 中，联合体（Union）是一种构造类型的数据结构。在一个联合体内，我们可以定义多种不同的数据类型，这些数据将会共享相同内存空间，这在一些需要复用内存的情况下，可以达到节省空间的目的。不过，根据 C++98 标准，并不是所有的数据类型都能够成为联合体的数据成员。我们先来看一段代码，如代码清单 3-38 所示。

代码清单 3-38

```
struct Student{
    Student(bool g, int a): gender(g), age(a){}
    bool gender;
    int age;
};

union T {
    Student s;  //  编译失败，不是一个 POD 类型
    int id;
    char name[10];
};

// 编译选项 :g++ -std=c++98 3-7-1.cpp
```

在代码清单 3-38 中，我们声明了一个 Student 的类型。根据在 3.6 节中学习到的知识，由于 Student 自定义了一个构造函数，所以该类型是非 POD 的。在 C++98 标准中，union T 是无法通过编译的。事实上，除了非 POD 类型之外，C++98 标准也不允许联合体拥有静态或引用类型的成员。这样虽然可能在一定程度上保证了和 C 的兼容性，不过也为联合体的使用带来了很大的限制。

而且通过长期的实践应用证明，C++98 标准对于联合体的限制是完全没有必要的。随着 C++ 的发展，C 与 C++ 在一些方面存在着事实的不同。比如典型的“复数”类型 complex，由于 C 语言中的复数遵从代码运行平台的二进制接口的规定，通常是一个平台上的内置类型。而在 C++ 中，复数则常常是一个模板来实现的。这样一来，形如下面这样的 C++ 声明，通常 C++98 编译器认为是不合法的。

```
union {
    double d;
    complex<double> cd;    // 错误，complex 不是一个 POD 类型
};
```

形如下面这样的 C 声明则被认为是合法的代码。

```
union {
    double d;
```

```
    complex cd;
};
```

这样一来，联合体保持与 C 一定程度上的兼容的特性也渐渐形同虚设。因此，在新的 C++11 标准中，取消了联合体对于数据成员类型的限制。标准规定，任何非引用类型都可以成为联合体的数据成员，这样的联合体即所谓的非受限联合体（Unrestricted Union）。所以如果使用 g++ 或者 clang++ 中的 -std=c++11 选项编译代码清单 3-38 的例子，是能够通过的。此外，联合体拥有静态成员（在非匿名联合体中）的限制，也在 C++11 新标准中被删除了。不过从实践中，我们发现 C++11 的规则不允许静态成员变量的存在（否则所有该类型的联合体将共享一个值）。而静态成员函数存在的唯一作用，大概就是为了返回一个常数，我们可以看看代码清单 3-39 所示的例子。

代码清单 3-39

```
#include <iostream>
using namespace std;

union T{ static long Get() { return 32; } };

int main(){
    cout << T::Get() << endl;
}

// 编译选项 :g++ -std=c++11 3-7-2.cpp
```

在代码清单 3-39 中，我们就定义了一个有静态成员函数的联合体。不过看起来这里的 union T 更像是一个作用域限制符，并没有太大的实用意义。

在 C++98 中，标准规定了联合体会自动对未在初始化成员列表中出现的成员赋默认初值。然而对于联合体而言，这种初始化常常会带来疑问，因为在任何时刻只有一个成员可以是有效的，如代码清单 3-40 所示。

代码清单 3-40

```
union T{
    int x;
    double d;
    char b[sizeof(double)];
};

T t = {0};    // 到底是初始化第一个成员还是所有成员呢？

// 编译选项 :g++ -std=c++98 -c 3-7-3.cpp
```

代码清单 3-40 中使用了花括号组成的初始化列表，试图将成员变量 x 初始化为零，即

整个联合体的数据 t 中低位的 4 字节被初始化为 0，然而实际上，t 所占的 8 个字节将全部被置 0。

而在 C++11 中，为了减少这样的疑问，标准会默认删除一些非受限联合体的默认函数。比如，非受限联合体有一个非 POD 的成员，而该非 POD 成员类型拥有有非平凡的构造函数，那么非受限联合体成员的默认构造函数将被编译器删除。其他的特殊成员函数，例如默认拷贝构造函数、拷贝赋值操作符以及析构函数等，也将遵从此规则。我们可以看看代码清单 3-41 所示的这个例子。

代码清单 3-41

```
#include <string>
using namespace std;

union T {
    string s;    // string 有非平凡的构造函数
    int n;
};

int main() {
    T t;         // 构造失败，因为 T 的构造函数被删除了
}

// 编译选项 :g++ -std=c++11 3-7-4.cpp
```

在代码清单 3-41 中，联合体 T 拥有一个非 POD 的成员 s。而 string 有非平凡的构造函数，因此 T 的构造函数被删除，其类型的变量 t 也就无法声明成功。解决这个问题的办法是，由程序员自己为非受限联合体定义构造函数。通常情况下，placement new 会发挥很好的作用，如代码清单 3-42 所示。

代码清单 3-42

```
#include <string>
using namespace std;

union T {
    string s;
    int n;
public:
    // 自定义构造函数和析构函数
    T(){ new (&s) string; }
    ~T() { s.~string(); }
};

int main() {
    T t;          // 构造析构成功
}
```

```
// 编译选项 :g++ -std=c++11 3-7-5.cpp
```

在代码清单 3-42 中，我们定义了 union T 的构造和析构函数。构造时，采用 placement new 将 s 构造在其地址 &s 上。这里 placement new 的唯一作用只是调用了一下 string 的构造函数。而在析构时，又调用了 string 的析构函数。读者必须注意的是，析构的时候 union T 也必须是一个 string 对象，否则可能导致析构的错误（或者让析构函数为空，至少不会造成运行时错误）。这样一来，变量 t 的声明就可以通过编译了。

匿名非受限联合体可以运用于类的声明中，这样的类也被称为“枚举式的类”（union-like class）。我们可以看看代码清单 3-43 所示的例子。

代码清单 3-43

```
#include <cstring>
using namespace std;

struct Student{
    Student(bool g, int a): gender(g), age(a){}
    bool gender;
    int age;
};

class Singer {
public:
    enum Type {STUDENT, NATIVE, FOREIGNER};

    Singer(bool g, int a): s(g, a) { t = STUDENT; }
    Singer(int i): id(i) { t = NATIVE; }
    Singer(const char* n, int s) {
        int size = (s > 9) ? 9 : s;
        memcpy(name, n, size);
        name[s] = '\0';
        t = FOREIGNER;
    }

    ~Singer() {}

private:
    Type t;
    union {      // 匿名的非受限联合体
        Student s;
        int id;
        char name[10];
    };
};

int main(){
```

```
    Singer(true, 13);
    Singer(310217);
    Singer("J Michael", 9);
}

// 编译选项 :g++ -std=c++11 3-7-6.cpp
```

在代码清单 3-43 中，我们也把匿名非受限联合体成为类 Singer 的“变长成员”（variant member）。可以看到，这样的变长成员给类的编写带来了更大的灵活性，这是 C++98 标准中无法达到的。

3.8 用户自定义字面量

☞ **类别：部分人**

在 C/C++ 程序中，程序员常常会使用结构体或者类来创造新的类型，以满足实际的需求。比如在进行科学计算时，用户可能需要用到复数（通常会包含实部和虚部两部分）。而对于颜色，用户通常会需要一个四元组（三原色及 Alpha）。而对于奥运会组委会，他们则常常会需要七元组（标示来自七大洲的状况）。不过，有的时候自定义类型也会有些书写的麻烦，尤其是用户想声明一个自定义类型的“字面量”（literal）的时候。我们可以看看代码清单 3-44 所示的例子。

代码清单 3-44

```
#include <iostream>
using namespace std;

typedef unsigned char uint8;
struct RGBA{
    uint8 r;
    uint8 g;
    uint8 b;
    uint8 a;

    RGBA(uint8 R, uint8 G, uint8 B, uint8 A = 0):
        r(R), g(G), b(B), a(A){}
};

std::ostream & operator << (std::ostream & out, RGBA & col) {
    return out << "r: " << (int)col.r
        << ", g: " << (int)col.g
        << ", b: " << (int)col.b
        << ", a: " << (int)col.a << endl;
}
void blend(RGBA & col1, RGBA & col2){
    cout << "blend " << endl << col1 << col2 << endl;
```

```
}

int main() {
    RGBA col1(255, 240, 155);
    RGBA col2({15, 255, 10, 7});   // C++11 风格的列表初始化
    blend(col1, col2);
}

// 编译选项 :g++ -std=c++11 3-8-1.cpp
```

在代码清单 3-44 所示的例子中，我们在 main 函数中想对两个确定的 RGBA 变量进行运算。这里我们采用了传统的方式，即先声明两个 RGBA 的变量，并且赋予相应初值，再将其传给函数 blend。程序员在编写测试用例的时候，常会遇到需要声明较多值确定的 RGBA 变量。那么这样的声明变量 – 传值运算的方式是件非常麻烦的。如果自定义类型可以像内置类型一样向函数传递字面常量，比如向函数 func 传递字面常量 func(2, 5.0f)，无疑这样的测试代码会简化很多。

C++11 标准允许了这种想象，即我们可以通过定一个后缀标识的操作符，将声明了该后缀标识的字面量转化为需要的类型。我们可以看一看代码清单 3-45 所示的代码。

代码清单 3-45

```
#include <cstdlib>
#include <iostream>
using namespace std;

typedef unsigned char uint8;
struct RGBA{
    uint8 r;
    uint8 g;
    uint8 b;
    uint8 a;
    RGBA(uint8 R, uint8 G, uint8 B, uint8 A = 0):
        r(R), g(G), b(B), a(A){}
};

RGBA operator "" _C(const char* col, size_t n) {     // 一个长度为 n 的字符串 col
    const char* p = col;
    const char* end = col + n;
    const char* r, *g, *b, *a;
    r = g = b = a = nullptr;

    for(; p != end; ++p) {
        if (*p == 'r') r = p;
        else if (*p == 'g') g = p;
        else if (*p == 'b') b = p;
        else if (*p == 'a') a = p;
```

```
    }

    if ((r == nullptr) || (g == nullptr) || (b == nullptr))
        throw;
    else if (a == nullptr)
        return RGBA(atoi(r+1), atoi(g+1), atoi(b+1));
    else
        return RGBA(atoi(r+1), atoi(g+1), atoi(b+1), atoi(b+1));
}

std::ostream & operator << (std::ostream & out, RGBA & col) {
    return out << "r: " << (int)col.r
        << ", g: " << (int)col.g
        << ", b: " << (int)col.b
        << ", a: " << (int)col.a << endl;
}

void blend(RGBA && col1, RGBA && col2) {
    // Some color blending action
    cout << "blend " << endl << col1 << col2 << endl;
}

int main() {
    blend("r255 g240 b155"_C, "r15 g255 b10 a7"_C);
}

// 编译选项 :g++ -std=c++11 3-8-2.cpp
```

这里，我们声明了一个字面量操作符（literal operator）函数：RGBA operator "" _C(const char* col, size_t n) 函数。这个函数会解析以 _C 为后缀的字符串，并返回一个 RGBA 的临时变量。有了这样一个用户字面常量的定义，main 函数中我们不再需要通过声明 RGBA 类型的声明变量 – 传值运算的方式来传递实际意义上的常量。通过声明一个字符串以及一个 _C 后缀，operator "" _C 函数会产生临时变量。blend 函数就可以通过右值引用获得这些临时值并进行计算了。这样一来，用户就完成了定义自定义类型的字面常量，main 函数中的代码书写显得更加清晰。

除去字符串外，后缀声明也可以作用于数值，比如，用户可能使用 60W、120W 的表示方式来标识功率，用 50kg 来表示质量，用 1200N 来表示力等。请看代码清单 3-46 所示的例子。

代码清单 3-46

```
struct Watt{ unsigned int v; };

Watt operator "" _w(unsigned long long v) {
    return {(unsigned int)v};
}
```

```
int main() {
    Watt capacity = 1024_w;
}

// 编译选项 :g++ -std=c++11 3-8-3.cpp
```

这里我们用 _w 后缀来标识瓦特。除了整型数，用户自定义字面量还可以用于浮点型数等的字面量。不过必须注意的是，在 C++11 中，标准要求声明字面量操作符有一定的规则，该规则跟字面量的“类型”密切相关。C++11 中具体规则如下：

- ❑如果字面量为整型数，那么字面量操作符函数只可接受 unsigned long long 或者 const char* 为其参数。当 unsigned long long 无法容纳该字面量的时候，编译器会自动将该字面量转化为以 \0 为结束符的字符串，并调用以 const char* 为参数的版本进行处理。
- ❑如果字面量为浮点型数，则字面量操作符函数只可接受 long double 或者 const char* 为参数。const char* 版本的调用规则同整型的一样（过长则使用 const char* 版本）。
- ❑如果字面量为字符串，则字面量操作符函数函数只可接受 const char*, size_t 为参数（已知长度的字符串）。
- ❑如果字面量为字符，则字面量操作符函数只可接受一个 char 为参数。

总体上来说，用户自定义字面量为代码书写带来了极大的便利。此外，在使用这个特性的时候，应该注意以下几点：

- ❑在字面量操作符函数的声明中，operator "" 与用户自定义后缀之间必须有空格。
- ❑后缀建议以下划线开始。不宜使用非下划线后缀的用户自定义字符串常量，否则会被编译器警告。当然，这也很好理解，因为如果重用形如 201203L 这样的字面量，后缀“L”无疑会引起一些混乱的状况。为了避免混乱，程序员最好只使用下划线开始的后缀名。

3.9 内联名字空间

☞ **类别：部分人**

在老式的 C 语言编程的实际项目中，我们常会需要一个“字典”来记录程序中所有的名字。这是由于 C 中所有的非静态全局变量、非静态的函数名都是都是全局共享的。那么对于多个程序员合作编程而言，总是需要知道自己给变量函数取的名字是否冲突，以避免发生编译错误，因此字典是一种使用 C 语言合作编程的一种重要交流手段。

在 C++ 中，引入了名字空间（namespace）这样一个概念。名字空间的目的是分割全局共享的名字空间。程序员在编写程序时可以建立自己的名字空间，而使用者则可以通过双冒号“空间名 :: 函数 / 变量名”的形式来引用自己需要的版本。这就解决了 C 中名字冲突的问题。不过有很多时候，我们会遇到一个名字空间下包含多个子名字空间的状况。子名字空间

通常会带来一些使用上的不便。我们来看看代码清单 3-47 所示的这个例子。

代码清单 3-47

```
#include <iostream>
using namespace std;
// 这是 Jim 编写的库，用了 Jim 这个名字空间
namespace Jim {
    namespace Basic {
        struct Knife{ Knife() { cout << "Knife in Basic." << endl; } };
        class CorkScrew{};
    }
    namespace Toolkit {
        template<typename T> class SwissArmyKnife{};
    }
    // ...
    namespace Other {
        Knife b;              // 无法通过编译
        struct Knife{ Knife() { cout << "Knife in Other" << endl;} };
        Knife c;              // Knife in Other
        Basic::Knife k;       // Knife in Basic
    }
}

// 这是 LiLei 在使用 Jim 的库
using namespace Jim;
int main() {
    Toolkit::SwissArmyKnife<Basic::Knife> sknife;
}

// 编译选项 :g++ 3-9-1.cpp
```

在代码清单 3-47 中，库的编写者 Jim 用名字空间将自己的代码封装起来。同时，该程序员把名字空间继续细分为 Basic、Toolkit 及 Other 等几个。可以看到，通过名字空间的细分，Other 名字空间中不能直接引用 Basic 名字空间中的名字 Knife。而 Other 名字空间中定义了 Knife 类型，那么变量 c 的声明就会导致其使用的 Knife 类型是属于名字空间 Other 中的版本的。这样的使用名字空间的方式是非常清楚的。

不过 Jim 这样会带来一个问题，即库的使用者在使用 Jim 名字空间的时候，需要知道太多的子名字空间的名字。使用者显然不希望声明一个 sknife 变量时，需要 Toolkit::SwissArmyKnife<Basic::Knife> 这么长的类型声明。而从库的提供者 Jim 的角度看，通常也没必要让使用者 LiLei 看到子名字空间，因此他可能考虑这样修改代码，如代码清单 3-48 所示。

代码清单 3-48

```
#include <iostream>
using namespace std;
```

```
namespace Jim {
    namespace Basic {
        struct Knife{ Knife() { cout << "Knife in Basic." << endl; } };
        class CorkScrew{};
    }
    namespace Toolkit{
        template<typename T> class SwissArmyKnife{};
    }
    // ...
    namespace Other{
        // ...
    }
    // 打开一些内部名字空间
    using namespace Basic;
    using namespace Toolkit;
}

// LiLei 决定对该 class 进行特化
namespace Jim {
    template<> class SwissArmyKnife<Knife>{};    // 编译失败
}

using namespace Jim;
int main() {
    SwissArmyKnife<Knife> sknife;
}

// 编译选项 :g++ 3-9-2.cpp
```

在代码清单 3-48 所示的例子中，Jim 在名字空间 Jim 的最后部分，打开了（using）Basic 和 Toolkit 两个名字空间（我们省略了关于 Other 名字空间中的部分）。这样一来在代码清单 3-48 中遇到的名字过长的问题就不复存在了。不过这里又有了新的问题：库的使用者 LiLei 由于觉得 Toolkit 中的模板 SwissArmyKnife 有的时候不太合用，所以决定特化一个 SwissArmyKnife<Knife> 的版本。这个时候，我们编译该例子则会失败。这是由于 C++98 标准不允许在不同的名字空间中对模板进行特化造成的。

在 C++11 中，标准引入了一个新特性，叫做“内联的名字空间”。通过关键字“inline namespace”就可以声明一个内联的名字空间。内联的名字空间允许程序员在父名字空间定义或特化子名字空间的模板。我们可以看看代码清单 3-49 所示的例子。

代码清单 3-49

```
#include <iostream>
using namespace std;

namespace Jim {
```

```
    inline namespace Basic {
        struct Knife{ Knife() { cout << "Knife in Basic." << endl; } };
        class CorkScrew{};
    }
    inline namespace Toolkit {
        template<typename T> class SwissArmyKnife{};
    }
    // ...
    namespace Other{
        Knife b;              // Knife in Basic
        struct Knife{ Knife() { cout << "Knife in Other" << endl;} };
        Knife c;              // Knife in Other
        Basic::Knife k;       // Knife in Basic
    }
}

// 这是 LiLei 在使用 Jim 的库
namespace Jim {
    template<> class SwissArmyKnife<Knife>{};    // 编译通过
}

using namespace Jim;
int main() {
    SwissArmyKnife<Knife> sknife;
}

// 编译选项 :g++ -std=c++11 3-9-3.cpp
```

代码清单 3-49 中，我们将名字空间 Basic 和 Toolkit 都声明为 inline 的。此时，LiLei 对库中模板的偏特化（SwissArmyKnife<Knife>）则可以通过编译。不过这里我们需要再次注意一下 Other 这个名字空间中的状况。可以看到，变量 b 的声明语句是可以通过编译的，而且其被声明为一个 Basic::Knife 的类型。如果换个角度理解的话，在子名字空间 Basic 中的名字现在看起来就跟在父名字空间 Jim 中一样。了解了这一点，读者或者会皱起眉头，Jim 名字空间中的良好分隔明显被破坏了，要做到这样的效果，只需要把 Knife 和 CorkScrew 放到全局名字空间中就可以了，根本不用 inline namespace 这么复杂。事实上，这跟 inline namespace 的使用方式有关。我们可以看看代码清单 3-50 所示的例子。

代码清单 3-50

```
#include <iostream>
using namespace std;

namespace Jim {
#if __cplusplus == 201103L
    inline
#endif
    namespace cpp11{
```

```
        struct Knife{ Knife() { cout << "Knife in c++11." << endl; } };
        // ...
    }

#if __cplusplus < 201103L
    inline
#endif
    namespace oldcpp{
        struct Knife{ Knife() { cout << "Knife in old c++." << endl; } };
        // ...
    }
}

using namespace Jim;
int main() {
    Knife a;            // Knife in c++11. (默认版本)
    cpp11::Knife b;     // Knife in c++11. (强制使用 cpp11 版本)
    oldcpp::Knife c;    // Knife in old c++. (强制使用 oldcpp11 版本)
}

// 编译选项 :g++ -std=c++11 3-9-4.cpp
```

在代码清单 3-50 中，Jim 为它的名字空间设定了两个子名字空间：cpp11 和 oldcpp。这里我们看到了在 2.1 节中提到的关于 C++ 的宏 __cplusplus。代码的意思是，如果现在的宏 __cplusplus 等于 201103 这个常数，那么就将名字空间 cpp11 内联到 Jim 中，而如果小于 201103，则将名字空间 oldcpp 内联到 Jim 中。这样一来，编译器就可以根据当前编译器对 C++ 支持的状况，选择合适的实现版本。而如果需要的话，我们依然可以通过名字空间的方式（如 cpp11::Knife）来访问相应名字空间中的类型、数据、函数等。这对程序库的发布很有好处，因为需要长期维护的程序库，可能版本间的接口和实现等都随着程序库的发展而发生了变化。那么根据需要将合适的名字空间导入到父名字空间中，无疑会方便库的使用。

事实上，在 C++ 标准程序库中，开发者已经开始这么做了。如果程序员需要长期维护、发布不同库的版本，不妨试用一下内联名字空间这个特性。

还有一点需要指出的是，匿名的名字空间同样可以把其包含的名字导入父名字空间。所以读者可能认为代码清单 3-50 中的代码同样可以通过匿名名字空间与宏组合来实现。不过跟代码清单 3-48 中使用 using 打开名字空间的情况一样，匿名名字空间无法允许在父名字空间的模板特化。这也是 C++11 中为什么要引入新的内联名字空间的一个根本原因。不过与我们在代码清单 3-49 中看到的一样，名字空间的内联会破坏该名字空间本身具有的封装性，所以程序员不应该在需要隔离名字的时候使用 inline namespace 关键字。

此外，在代码实践时，读者可能还会被一些 C++ 的语言特性迷惑，比较典型的是所谓“参数关联名称查找”，即 ADL（Argument-Dependent name Lookup）。这个特性允许编译器在名字空间内找不到函数名称的时候，在参数的名字空间内查找函数名字。比如说下面这个例子：

```
namespace ns_adl{
    struct A{};
    void ADLFunc(A a){} // ADLFunc 定义在 namespace ns_adl 中
}

int main() {
    ns_adl::A a;
    ADLFunc(a); // ADLFunc 无需声明名字空间
}
```

函数 ADLFunc 就无需在使用时声明其所在的名字空间，因为编译器可以在其参数 a 的名字空间 ns_adl 中找到 ADLFunc，编译也就不会报错了。

ADL 带来了一些使用上的便利性，不过也在一定程度上破坏了 namespace 的封装性。很多人认为使用 ADL 会带来极大的负面影响[⊖]。因此，比较好的使用方式，还是在使用任何名字前打开名字空间，或者使用“::”列出变量、函数完整的名字空间。

3.10 模板的别名

☞ **类别：部分人**

在 C++ 中，使用 typedef 为类型定义别名。比如：

```
typedef int myint;
```

就定义了一个 int 的别名 myint。当遇到一些比较长的名字，尤其是在使用模板和域的时候，使用别名的优势会更加明显。比如：

```
typedef std::vector<std::string> strvec;
```

这里使用 strvec 作为 std::vector<std::string> 的别名。在 C++11 中，定义别名已经不再是 typedef 的专属能力，使用 using 同样也可以定义类型的别名，而且从语言能力上看，using 丝毫不比 typedef 逊色。我们可以看看代码清单 3-51 所示的这个例子。

代码清单 3-51

```
#include <iostream>
#include <type_traits>
using namespace std;

using uint = unsigned int;
typedef unsigned int UINT;
using sint = int;

int main() {
    cout << is_same<uint, UINT>::value << endl; // 1
```

⊖ 读者可以参考以下网页：http://stackoverflow.com/questions/2958648/what-are-the-pitfalls-of-adl

```
}

// 编译选项 :g++ -std=c++11 3-10-1.cpp
```

在本例中，使用了 C++11 标准库中的 is_same 模板类来帮助我们判断两个类型是否一致。is_same 模板类接受两个类型作为模板实例化时的参数，而其成员类型 value 则表示两个参数类型是否一样。在代码清单 3-51 所示的例子中我们看到，使用 using uint = unsigned int; 定义的类型别名 uint 和使用 typedef unsigned int UINT; 定义的类型别名 UINT 都是一样的类型。两者效果相同，或者说，在 C++11 中，using 关键字的能力已经包括了 typedef 的部分。

在使用模板编程的时候，using 的语法甚至比 typedef 更加灵活。比如下面这个例子：

```
template<typename T> using MapString = std::map<T, char*>;
MapString<int> numberedString;
```

在这里，我们“模板式”地使用了 using 关键字，将 std::map<T, char*> 定义为了一个 MapString 类型，之后我们还可以使用类型参数对 MapString 进行类型的实例化，而使用 typedef 将无法达到这样的效果。

3.11 一般化的 SFINEA 规则

☞ **类别：库作者**

在 C++ 模板中，有一条著名的规则，即 SFINEA - Substitution failure is not an error，中文直译即是“匹配失败不是错误”。更为确切地说，这条规则表示的是对重载的模板的参数进行展开的时候，如果展开导致了一些类型不匹配，编译器并不会报错。我们可以具体地看看代码清单 3-52 所示的来自 wikipedia 的例子[⊖]。

代码清单 3-52

```
struct Test {
    typedef int foo;
};

template <typename T>
void f(typename T::foo) {}  // 第一个模板定义 - #1

template <typename T>
void f(T) {}                // 第二个模板定义 - #2

int main() {
    f<Test>(10);     // 调用 #1.
    f<int>(10);      // 调用 #2. 由于 SFINEA，虽然不存在类型 int::foo，也不会发生编译错误
}
```

⊖ http://en.wikipedia.org/wiki/Substitution_failure_is_not_an_error

```
// 编译选项：g++ 2-14-1.cpp
```

在代码清单 3-52 中，我们重载了函数模板 f 的定义。第一个模板 f 接受的参数类型为 T::foo，这里我们通过 typename 来使编译器知道 T::foo 是一个类型。而第二个模板定义则接受一个 T 类型的参数。在 main 函数中，分别使用 f<Test> 和 f<int> 对模板进行实例化的时候会发现，对于 f<int> 来讲，虽然不存在 int::foo 这样的类型，编译器依然不会报错，相反地，编译器会找到第二个模板定义并对其进行实例化。这样一来，就保证了编译的正确性。

事实上，通过上面的例子我们可以发现，SFINAE 规则的作用比起其拗口的定义而言更为直观。基本上，这是一个使得 C++ 模板推导规则符合程序员想象的规则。通过 SFINAE，我们能够使得模板匹配更为“精确”，即使得一些模板函数、模板类在实例化时使用特殊的模板版本，而另外一些则使用通用的版本，这样就大大增加了模板设计使用上的灵活性。而在现实生活中，这样的使用方式在标准库中使用非常普遍（当你在标准库中发现一大堆的 __enable_if，或者应该想起这是 SFINAE）。因此也可以说，SFINAE 也是 C++ 模板推导中非常基础的一个特性。

不过在 C++98 中，标准对于 SFINAE 并没有完全清晰的描述，一些在模板参数中使用表达式的情况，并没有被主流编译器支持。我们可以看看代码清单 3-53 所示的这个来自于 C++11 标准提案中的例子。

代码清单 3-53

```
template <int I> struct A {};

char xxx(int);
char xxx(float);

template <class T> A<sizeof(xxx((T)0))> f(T){}

int main() {
    f(1);
}
// 编译选项：g++ -std=c++11
```

在代码清单 3-53 中，我们在定义函数模板 f 的时候，其返回值则定义为一个以 sizeof(xxx((T)0)) 为参数的类模板 A。这里值得注意的是，我们使用了 sizeof 表达式，以及强制的类型转换。事实上，这样的表达式是可以在模板实例化时被计算出的。不过由于实现上的复杂性，以及标准中并未明确地提及，大多数 C++98 编译器都会报出一个 SFINEA 失败信息。而事实上，这样灵活的用法却非常有用，比如本例中，程序员可以根据参数的长度而定义出不同返回值的模板函数 f（一种是 sizeof((int)0)，一种则是 sizeof((float)0)）。如果编译器拒绝了这样的使用方式，无疑会为泛型编程的应用带来一些限制。

在 C++11 中，标准对这样的状况，尤其是模板参数替换时使用了表达式的情况进行了明确规定，即表达式中没有出现“外部于表达式本身”的元素，比如说发生一些模板的实例化，或者隐式地产生一些拷贝构造函数的话，这样的模板推导都不会产生 SFINAE 失败（即编译器报错）。这样一来，C++11 中的一些新特性（比如我们将在第 4 章中讲到的 decltype 等）将能够成功地进行广泛的应用，进一步地，新的 STL 也将因此受益。

3.12 本章小结

在本章中，我们介绍了 C++11 中共 11 个崭新特性。这些新特性都在着重于通用性的考量下，经过标准委员会反复揣摩而最终成型。

最为引人注目的就是右值引用。右值引用堪称是 C++11 中的一项重大的变革。这次的变革，是以暴露原本一直被 C/C++ 掩盖得较好的左值右值关系为代价的。不过客观地讲，也是在程序员对 C++ 性能的一再紧逼下，左值右值的概念才终于“露出真身”。相比于 C++ 的其他概念，右值引用的理解会稍微复杂一些，但其目的却比较明确，就是实现所谓的移动语义。移动语义与在 C++98 中常见的拷贝语义在类的构造上采用了完全不同的方式。移动语义主要是将行将被释放的资源“偷”出来，作为行将构造的类型的资源。那么这势必就会跟变量生命期产生关系，跟右值、临时量打上交道。而最终，C++11 中又采用了右值引用的方式使得移动构造函数能够有效地获得这些右值临时量，以使程序员能够完成行为良好的移动语义。通过这样的移动语义，库的实现者可以巧妙地将各种形如堆内存的资源放入对象中，而不必担心在诸如函数传递的过程中带来过大的资源释放、申请开销。此外，标准制定者还趁机利用了右值引用来实现了所谓的完美转发，从技术上讲，完美转发就是通过引用折叠规则和模板推导规则，使得转发函数在不损失任何数据属性的情况下，将数据完美地传递给其他函数。完美转发在标准库中被广泛地使用，也是泛型编程中一种很好的包装方式。

在 C++11 中，标准则又重新回答了最基本的问题——什么是简单的类型及什么是复杂的类型，即怎么才算得上是 POD。C++11 的 POD 的概念分为平凡的和标准布局两个概念。标准布局强调了类型的数据在排布上是简单的，比如可以通过 memcpy 拷贝的简单类型。而平凡的则强调了类型没有复杂的构造、析构或者多态等看起来“不平凡”的行为。了解 POD 实际为了解 C++11 中其他很多的概念奠定了基础。

另外一个新引入的改动则是列表初始化。相比于 C++98 中的赋值表达式和值初始化，列表初始化主要被实现为标准库中的 initializer_list，使得用户不仅可以列表式地初始化内置类型、数组、STL 容器，还可以对自定义类型进行列表初始化。这应该是 C++ 标准中第一次出现与库实现结合得如此紧密的语言特性。而列表初始化相对于老式的初始化，对总是容易出错的类型收窄做了限制。总的说来，无论从使用的方便性还是安全性上讲，列表初始化都表现出了优良的特性，也是 C++ 核心语言进步的一种体现。

两种新的构造函数的声明方式——继承构造函数和委派构造函数，则都使得程序员能够

在编写构造函数中少写一些代码。前者我们将关注点放在了继承结构下使用 using 关键字将构造函数继承上，而后者，我们则把目光放在了单类型中多个构造函数间通过初始化列表的相互委托关系上。而用户自定义字面量将 C++ 中的各种重载再一次强化。通过后缀，用户可以将程序中非内置的几乎所有的字面量据为己用，产生自己的类型。同样地，这会减少 C++11 代码的书写量。另外一个简化，则是 using 的“泛化”，using 关键字已经能够像 typedef 一样定义别名，且其在模板中使用起来更佳。

此外，我们还介绍了能够避免意外的显式类型转换，以及能为类产生变长成员的非受限联合体。内联的名字空间则是用于库发布的一个特性，通过将子名字空间的名字导入父名字空间，用户可以方便地使用名字空间的名字。不过名字空间的内联也导致名字空间对名字封装的失效。因此通常情况下，这个特性都会结合宏一起使用。

最后我们还介绍了 SFINAE 规则的改进，通过这样的改进，C++11 进一步提高了范型编程的能力。

读完这一章，相信读者已经能够体会到 C++11 的一些风格。通用手段为先，如果能够使用更为通用的方法解决的话，就不再劳烦语言核心进行繁复的更改（这在列表初始化上体现得非常明显）。而在 C++11 中，我们也发现范型编程的力量越来越强大，语言层面的更改，很多都是在支持解决泛型编程中出现的一些问题（比如完美转发、模板别名、SFINAE 等）。如果熟悉了这些特性，那么必然程序代码会越写越少，性能也越来越好。

第④章

新手易学，老兵易用

在 C++ 变得原来越强大的同时，一些程序员也在抱怨使用 C++ 进行编码过于复杂。由于 STL 与语言核心部分的关联越来越紧密，而 STL 往往需要涉及解释一些模板的知识，很多新手也会因此对使用 C++ 进行编程产生排斥。C++11 的设计者对此作了改进。本章中，我们可以看到 auto 类型推导、基于范围的 for 循环等非常易用的特性。这些特性都非常具有亲和力，而用于进行代码编写也能有效地提升代码效率，必然会被 C++11 代码大量使用。

4.1 右尖括号 > 的改进

☞ **类别：所有人**

在 C++98 中，有一条需要程序员规避的规则：如果在实例化模板的时候出现了连续的两个右尖括号 >，那么它们之间需要一个空格来进行分隔，以避免发生编译时的错误。我们可以看代码清单 4-1 所示的例子。

代码清单 4-1

```
template <int i> class X{};
template <class T> class Y{};

Y<X<1> > x1;     // 编译成功
Y<X<2>> x2;      // 编译失败

// 编译选项 :g++ -c 4-1-1.cpp
```

在代码清单 4-1 中，我们定义了两个模板类型 X 和 Y，并且使用模板定义分别声明了以 X<1> 为参数的 Y<X<1> > 类型变量 x1，以及以 X<2> 为参数的 Y<X<2> > 类型变量 x2。不过 x2 的定义编译器却不能正确解析。在 x2 的定义中，编译器会把 >> 优先解析为右移符号。使用 gcc 编译的时候，我们会得到如下错误提示：

```
4-1-1.cpp:5:9: error: 'x2' was not declared in this scope
 Y<X<2>> x2;      // 编译失败
         ^
4-1-1.cpp:5:9: error: template argument 1 is invalid
4-1-1.cpp:5:3: error: template argument 1 is invalid
 Y<X<2>> x2;      // 编译失败
```

事实上，除去嵌套的模板标识，在使用形如 static_cast、dynamic_cast、reinterpret_cast，或者 const_cast 表达式进行转换的时候，我们也常会遇到相同的情况。

```
const vector<int> v = static_cast<vector<int>>(v);      // 无法通过编译
```

C++98 同样会将 >> 优先解析为右移。C++11 中，这种限制被取消了。事实上，C++11 标准要求编译器智能地去判断在哪些情况下 >> 不是右移符号。使用 C++11 标准，代码清单 4-1 所示代码则会成功地通过编译。

不过这些“智能”的判断也会带来一些与 C++98 的有趣的不兼容性。比如用户只是想让 >> 在模板的实例化中表示的是真正的右移，但是 C++11 会把它解析为模板参数界定符。比如代码清单 4-2 所示的代码。

代码清单 4-2

```
template <int i> class X {};
X<1 >> 5> x ;
```

如果使用 C++98 标准进行编译的话，这个例子会编译通过，因为编译器认为 X<1 >> 5> x ; 中的双尖括号是一个位移操作，那么最终可以得到一个形如 X<0> x 的模板实例。而如果使用 C++11 标准进行编译，那么程序员会得到一个编译错误的警告，因为编译器优先将双尖括号中的第一个 > 与 X 之后的 < 进行了配对。

虽然很少有人在模板实例化时同时进行位移操作，但是从语法上来说，C++98 和 C++11 确实在这一点上不兼容。要避免这样的不兼容性也很简单，使用圆括号将“1 >> 5”括起来，保证右移操作优先，就不会出现类似问题了。

4.2 auto 类型推导

☞ **类别：所有人**

4.2.1 静态类型、动态类型与类型推导

在编程语言的分类中，C/C++ 常被冠以“静态类型”的称号。而有的编程语言则号称是“动态类型”的，比如 Python。通常情况下，“静”和“动”的区别非常直观。我们可以看看下面这段简单的 Python 代码：

```
name = 'world\n'
print 'hello, ' % name
```

这段代码是 Python 中的一个 helloworld 的实现。这里的代码使用了一个变量 name，用于存储 world 这个字符串。接下来代码又使用 print 将 'hello, ' 字符串及 name 变量一起打印出来。请读者忽略其他的细节，只注意一下变量 name 的使用方式。事实上，变量 name 在使用

前没有进行过任何声明，而当程序员想使用时，可以拿来就用。

这种变量的使用方式显得非常随性，而在 C/C++ 程序员的眼中，每个变量使用前必须定义几乎是天经地义的事，这样通常被视为编程语言中的“静态类型”的体现。而对于如 Python、Perl、JavaScript 等语言中变量不需要声明，而几乎“拿来就用”的变量使用方式，则被视为是编程语言中“动态类型”的体现。不过从技术上严格地讲，静态类型和动态类型的主要区别在于对变量进行类型检查的时间点。对于所谓的静态类型，类型检查主要发生在编译阶段；而对于动态类型，类型检查主要发生在运行阶段。形如 Python 等语言中变量“拿来就用”的特性，则需要归功于一个技术，即类型推导。

事实上，类型推导也可以用于静态类型的语言中。比如在上面的 Python 代码中，如果按照 C/C++ 程序员的思考方式，world\n 表达式应该返回一个临时的字符串，所以即使 name 没有进行声明，我们也能轻松地推导出 name 的类型应该是一个字符串类型。在 C++11 中，这个想法得到了实现。C++11 中类型推导的实现的方式之一就是重定义了 auto 关键字。另外一个现实是 decltype，关于 decltype 的实现，我们将在后面详细描述。

我们可以使用 C++11 的方式书写一下刚才的 Python 代码，如代码清单 4-3 所示。

代码清单 4-3

```
#include <iostream>
using namespace std;

int main() {
    auto name = "world.\n";
    cout << "hello, " << name;
}

// 编译选项 :g++ -std=c++11 4-2-1.cpp
```

这里我们使用了 auto 关键字来要求编译器对变量 name 的类型进行自动推导。这里编译器根据它的初始化表达式的类型，推导出 name 的类型为 char*。这样一来就达到了跟刚才的 Python 代码差不多的效果。

事实上，auto 关键字在早期的 C/C++ 标准中有着完全不同的含义。声明时使用 auto 修饰的变量，按照早期 C/C++ 标准的解释，是具有自动存储期的局部变量。不过现实情况是该关键字几乎无人使用，因为一般函数内没有声明为 static 的变量总是具有自动存储期的局部变量。因此在 C++11 中，标准委员会决定赋予 auto 全新的含义，即 auto 不再是一个存储类型指示符（storage-class-specifier，如 static、extern、thread_local 等都是存储类型指示符），而是作为一个新的类型指示符（type-specifier，如 int、float 等都是类型指示符）来指示编译器，auto 声明变量的类型必须由编译器在编译时期推导而得。

我们可以通过代码清单 4-4 所示的例子来了解一下 auto 类型推导的基本用法。

代码清单 4-4

```
int main() {
    double foo();
    auto x = 1;        // x的类型为int
    auto y = foo();   // y的类型为double
    struct m { int i; }str;
    auto str1 = str;     // str1的类型是struct m

    auto z;       // 无法推导，无法通过编译
    z = x;
}

// 编译选项:g++ -std=c++11 4-2-2.cpp
```

在代码清单 4-4 中，变量 x 被初始化为 1，因为字面常量 1 的类型为 const int，所以编译器推导出 x 的类型应该为 int（这里 const 类型限制符被去掉了，后面会解释）。同理在变量 y 的定义中，auto 类型的 y 被推导为 double 类型；而在 auto str1 的定义中，其类型被推导为 struct m。

值得注意的是变量 z，这里我们使用 auto 关键字来“声明”z，但不立即对其进行定义，此时编译器则会报错。这跟通过其他关键字（除去引用类型的关键字）先声明后定义变量的使用规则是不同的。auto 声明的变量必须被初始化，以使编译器能够从其初始化表达式中推导出其类型。从这个意义上来讲，auto 并非一种“类型”声明，而是一个类型声明时的“占位符”，编译器在编译时期会将 auto 替代为变量实际的类型。

4.2.2 auto 的优势

直观地，auto 推导的一个最大优势就是在拥有初始化表达式的复杂类型变量声明时简化代码。由于 C++ 的发展，声明变量类型也变得越来越复杂，很多时候，名字空间、模板成为了类型的一部分，导致程序员在使用库的时候如履薄冰。我们可以看看代码清单 4-5 所示的例子。

代码清单 4-5

```
#include <string>
#include <vector>

void loopover(std::vector<std::string> & vs) {
    std::vector<std::string>::iterator i = vs.begin(); // 想要使用iterator，往往需要
                                                       书写大量代码

    for (; i < vs.end(); i++) {
        // 一些代码
    }
```

```
}

// 编译选项 :g++ -c 4-2-3.cpp
```

代码清单 4-5 中，我们在不使用 using namespace std 的情况下（事实上，很多专家的建议就是如此）想对一个 vector 数组进行循环。可以看到，当我们想定义一个迭代器 i 的时候，我们必须写出 std::vector<std::string>::iterator 这样长的类型声明。即使是一位擅长克服各种困难的 C++ 老手，也不会对如此冗长的代码视而不见。而使用 auto 的话，代码会的可读性可以成倍增长，如代码清单 4-6 所示。

代码清单 4-6

```
#include <string>
#include <vector>

void loopover(std::vector<std::string> & vs) {

    for (auto i = vs.begin(); i < vs.end(); i++) {
        // 一些代码
    }
}

// 编译选项 :g++ -c -std=c++11 4-2-4.cpp
```

如我们所见，使用了 auto，程序员甚至可以将 i 的声明放入 for 循环中，i 的类型将由表达式 vs.begin() 推导出。事实上，形如代码清单 4-5 的复杂声明在使用 STL 的代码中处处可见，在 C++11 中，由于 auto 的存在，使用 STL 将会变得更加容易，写出的代码也会更加清晰可读。

auto 的第二个优势则在于可以免除程序员在一些类型声明时的麻烦，或者避免一些在类型声明时的错误。事实上，在 C/C++ 中，存在着很多隐式或者用户自定义的类型转换规则（比如整型与字符型进行加法运算后，表达式返回的是整型，这是一条隐式规则）。这些规则并非很容易记忆，尤其是在用户自定义了很多操作符之后。而这个时候，auto 就有用武之地了。我们可以看看代码清单 4-7 所示的例子。

代码清单 4-7

```
class PI {
public:
    double operator* (float v) {
        return (double)val * v;     // 这里精度被扩展了
    }
    const float val = 3.1415927f;
};
```

```
int main() {
    float radius = 1.7e10;
    PI pi;
    auto circumference = 2 * (pi * radius);
}

// 编译选项 :g++ -std=c++11 4-2-5.cpp
```

代码清单 4-7 中，定义了 float 型的变量 radius（半径）以及一个自定义类型 PI 变量 pi（π 值），在计算圆周长的时候，使用了 auto 类型来定义变量 circumference。这里，PI 在与 float 类型数据相乘时，其返回值为 double。而 PI 的定义可能是在其他的地方（头文件里），main 函数的程序员可能不知道 PI 的作者为了避免数据上溢或者精度降低而返回了 double 类型的浮点数。因此 main 函数程序员如果使用 float 类型声明 circumference，就可能享受不了 PI 作者细心设计带来的好处。反之，将 circumference 声明为 auto，则毫无问题，因为编译器已经自动地做了最好的选择。

值得指出的是，auto 并不能解决所有的精度问题，我们可以看看代码清单 4-8 所示的例子。

代码清单 4-8

```
#include <iostream>
using namespace std;

int main() {
    unsigned int a = 4294967295;     // 最大的 unsigned int 值
    unsigned int b = 1;
    auto c = a + b;          // c 的类型依然是 unsigned int

    cout << "a = " << a << endl;     // a = 4294967295
    cout << "b = " << b << endl;     // b = 1
    cout << "a + b = " << c << endl;// a + b = 0
    return 0;
}

// 编译选项 :g++ -std=c++11 4-2-6.cpp
```

在代码清单 4-8 中，程序员可能指望通过声明变量 c 为 auto 就能解决 a + b 溢出的问题。而实际上由于 a+b 返回的依然是 unsigned int 的值，故而 c 的类型依然被推导为 unsigned int，auto 并不能帮上忙。这跟一些动态类型语言中数据会自动进行扩展的特性还是不一样的。

auto 的第三个优点就是其“自适应”性能够在一定程度上支持泛型的编程。

我们再回到代码清单 4-7 的例子，这里假设 PI 的作者改动了 PI 的定义，比如将 operator* 返回值变为 long double，此时，main 函数并不需要修改，因为 auto 会“自适应”

新的类型。

同理，对于不同的平台上的代码维护，auto 也会带来一些“泛型”的好处。这里我们以 strlen 函数为例，在 32 位的编译环境下，strlen 返回的为一个 4 字节的整型，而在 64 位的编译环境下，strlen 会返回一个 8 字节的整型。虽然系统库 <cstring> 为其提供了 size_t 类型来支持多平台间的代码共享支持，但是使用 auto 关键字我们同样可以达到代码跨平台的效果。

```
auto var = strlen("hello world!").
```

由于 size_t 的适用范围往往局限于 <cstring> 中定义的函数，auto 的适用范围明显更为广泛。

当 auto 应用于模板的定义中，其“自适应”性会得到更加充分的体现。我们可以看看代码清单 4-9 所示的例子。

代码清单 4-9

```
template<typename T1, typename T2>
double Sum(T1 & t1, T2 & t2) {
    auto s = t1 + t2;    // s 的类型会在模板实例化时被推导出来
    return s;
}

int main() {
    int a = 3;
    long b = 5;
    float c = 1.0f, d = 2.3f;

    auto e = Sum<int ,long>(a, b);        // s 的类型被推导为 long
    auto f = Sum<float,float>(c, d);      // s 的类型被推导为 float
}

// 编译选项 :g++ -std=c++11 4-2-7.cpp
```

在代码清单 4-9 中，Sum 模板函数接受两个参数。由于类型 T1、T2 要在模板实例化时才能确定，所以在 Sum 中将变量 s 的类型声明为 auto 的。在函数 main 中我们将模板实例化时，Sum<int, long> 中的 s 变量会被推导为 long 类型，而 Sum<float, float> 中的 s 变量则会被推导为 float。可以看到，auto 与模板一起使用时，其“自适应”特性能够加强 C++ 中“泛型”的能力。不过在这个例子中，由于总是返回 double 类型的数据，所以 Sum 模板函数的适用范围还是受到了一定的限制，在 4.4 节中，我们可以看到怎么使用追踪返回类型的函数声明来完全释放 Sum 泛型的能量。

另外，应用 auto 还会在一些情况下取得意想不到的好效果。我们可以看看代码清单 4-10 所示的例子[⊖]。

⊖ 本例素材来源于 GNU C 的手册 http://gcc.gnu.org/onlinedocs/gcc/Typeof.html。

代码清单 4-10

```
#define Max1(a, b) ((a) > (b)) ? (a) : (b)
#define Max2(a, b) ({ \
        auto _a = (a); \
        auto _b = (b); \
        (_a > _b) ? _a: _b; })

int main() {
    int m1 = Max1(1*2*3*4, 5+6+7+8);
    int m2 = Max2(1*2*3*4, 5+6+7+8);
}

// 编译选项 :g++ -std=c++11 4-2-8.cpp
```

在代码清单 4-10 中，我们定义了两种类型的宏 Max1 和 Max2。两者作用相同，都是求 a 和 b 中较大者并返回。前者采用传统的三元运算符表达式，这可能会带来一定的性能问题。因为 a 或者 b 在三元运算符中都出现了两次，那么无论是取 a 还是取 b，其中之一都会被运算两次。而在 Max2 中，我们将 a 和 b 都先算出来，再使用三元运算符进行比较，就不会存在这样的问题了。

在传统的 C++98 标准中，由于 a 和 b 的类型无法获得，所以我们无法定义 Max2 这样高性能的宏。而新的标准中的 auto 则提供了这种可行性。

4.2.3 auto 的使用细则

虽然我们在 4.2.1 节及 4.2.2 节中看到了很多关于 auto 的使用，不过 auto 使用上还有很多语法相关的细节。如果读者在使用 auto 的时候遇到一些不理解的状况，不妨回头来查阅一下这些规则。

首先我们可以看看 auto 类型指示符与指针和引用之间的关系。在 C++11 中，auto 可以与指针和引用结合起来使用，使用的效果基本上会符合 C/C++ 程序员的想象。我们可以看看代码清单 4-11 所示的例子。

代码清单 4-11

```
int x;
int * y = &x;
double foo();
int & bar();

auto * a = &x;       // int*
auto & b = x;        // int&
auto c = y;          // int*
auto * d = y;        // int*
auto * e = &foo();   // 编译失败，指针不能指向一个临时变量
```

```
auto & f = foo();    // 编译失败，nonconst 的左值引用不能和一个临时变量绑定
auto g = bar();      // int
auto & h = bar();    // int&

// 编译选项 :g++ -std=c++11 4-2-9.cpp
```

本例中，变量 a、c、d 的类型都是指针类型，且都指向变量 x。实际上对于 a、c、d 三个变量而言，声明其为 auto * 或 auto 并没有区别。而如果要使得 auto 声明的变量是另一个变量的引用，则必须使用 auto &，如同本例中的变量 b 和 h 一样。

其次，auto 与 volatile 和 const 之间也存在着一些相互的联系。volatile 和 const 代表了变量的两种不同的属性：易失的和常量的。在 C++ 标准中，它们常常被一起叫作 cv 限制符（cv-qualifier）。鉴于 cv 限制符的特殊性，C++11 标准规定 auto 可以与 cv 限制符一起使用，不过声明为 auto 的变量并不能从其初始化表达式中“带走”cv 限制符。我们可以看看代码清单 4-12 所示的例子。

代码清单 4-12

```
double foo();
float * bar();

const auto a = foo();          // a: const double
const auto & b = foo();        // b: const double&
volatile auto * c = bar();     // c: volatile float*

auto d = a;                    // d: double
auto & e = a;                  // e: const double &
auto f = c;                    // f: float *
volatile auto & g = c;         // g: volatile float * &

// 编译选项 :g++ -std=c++11 4-2-10.cpp
```

在代码清单 4-12 中，我们可以通过非 cv 限制的类型初始化一个 cv 限制的类型，如变量 a、b、c 所示。不过通过 auto 声明的变量 d、f 却无法带走 a 和 f 的常量性或者易失性。这里的例外还是引用，可以看出，声明为引用的变量 e、g 都保持了其引用的对象相同的属性（事实上，指针也是一样的）。

此外，跟其他的变量指示符一样，同一个赋值语句中，auto 可以用来声明多个变量的类型，不过这些变量的类型必须相同。如果这些变量的类型不相同，编译器则会报错。事实上，用 auto 来声明多个变量类型时，只有第一个变量用于 auto 的类型推导，然后推导出来的数据类型被作用于其他的变量。所以不允许这些变量的类型不相同，如代码清单 4-13 所示。

代码清单 4-13

```
auto x = 1, y = 2;          // x 和 y 的类型均为 int

// m 是一个指向 const int 类型变量的指针 , n 是一个 int 类型的变量
const auto* m = &x, n = 1;

auto i = 1, j = 3.14f;  // 编译失败

auto o = 1, &p = o, *q = &p;      // 从左向右推导

// 编译选项 :g++ -std=c++11 4-2-11.cpp -c
```

在代码清单 4-13 中，我们使用 auto 声明了两个类型相同变量 x 和 y，并用逗号进行分隔，这可以通过编译。而在声明变量 i 和 j 的时候，按照我们所说的第一变量用于推导类型的规则，那么由于 x 所推导出的类型是 int，那么对于变量 j 而言，其声明就变成了 int j = 3.14f，这无疑会导致精度的损失。而对于变量 m 和 n，就变得非常有趣，这里似乎是 auto 被替换成了 int，所以 m 是一个 int * 指针类型，而 n 只是一个 int 类型。同样的情况也发生在变量 o、p、q 上，这里 o 是一个类型为 int 的变量，p 是 o 的引用，而 q 是 p 的指针。auto 的类型推导按照从左往右，且类似于字面替换的方式进行。事实上，标准里称 auto 是一个将要推导出的类型的“占位符”（placeholder）。这样的规则无疑是直观而让人略感意外的。当然，为了不必要的繁琐记忆，程序员可以选择每一个 auto 变量的声明写成一行（有些观点也认为这是好的编程规范）。

此外，只要能够进行推导的地方，C++11 都为 auto 指定了详细的规则，保证编译器能够正确地推导出变量的类型。包括 C++11 新引入的初始化列表，以及 new，都可以使用 auto 关键字，如代码清单 4-14 所示。

代码清单 4-14

```
#include <initializer_list>

auto x = 1;
auto x1(1);

auto y {1};       // 使用初始化列表的 auto
auto z = new auto(1);    // 可以用于 new

// 编译选项 :g++ -std=c++11 -c 4-2-12.cpp
```

代码清单 4-14 中，auto 变量 y 的初始化使用了初始化列表，编译器可以保证 y 的类型推导为 int。而 z 指针所指向的堆变量在分配时依然选择让编译器对类型进行推导，同样的，编译器也能够保证这种方式下类型推导的正确性。

不过 auto 也不是万能的，受制于语法的二义性，或者是实现的困难性，auto 往往也会有使用上的限制。这些例外的情况都写在了代码清单 4-15 所示的例子当中。

代码清单 4-15

```
#include <vector>
using namespace std;

void fun(auto x =1){}  // 1: auto 函数参数，无法通过编译

struct str{
    auto var = 10;    // 2: auto 非静态成员变量，无法通过编译
};

int main() {
    char x[3];
    auto y = x;
    auto z[3] = x; // 3: auto 数组，无法通过编译

    // 4: auto 模板参数（实例化时），无法通过编译
    vector<auto> v = {1};
}

// 编译选项 :g++ -std=c++11 4-2-13.cpp
```

我们分别来看看代码清单 4-15 中的 4 种不能推导的情况。

1）对于函数 fun 来说，auto 不能是其形参类型。可能读者感觉对于 fun 来说，由于其有默认参数，所以应该推导 fun 形参 x 的类型为 int 型。但事实却无法符合大家的想象。因为 auto 是不能做形参的类型的。如果程序员需要泛型的参数，还是需要求助于模板。

2）对于结构体来说，非静态成员变量的类型不能是 auto 的。同样的，由于 var 定义了初始值，读者可能认为 auto 可以推导 str 成员 var 的类型为 int 的。但编译器阻止 auto 对结构体中的非静态成员进行推导，即使成员拥有初始值。

3）声明 auto 数组。我们可以看到，main 中的 x 是一个数组，y 的类型是可以推导的，而声明 auto z[3] 这样的数组同样会被编译器禁止。

4）在实例化模板的时候使用 auto 作为模板参数，如 main 中我们声明的 vector<auto> v。虽然读者可能认为这里一眼而知是 int 类型，但编译器却阻止了编译。

以上 4 种情况的特点基本相似，人为地观察很容易能够推导出 auto 所在位置应有的类型，但现有的 C++11 的标准还没有支持这样的使用方式。如果程序员遇到 auto 不够聪明的情况，不妨回头看看是否违背了以上一些规则。

此外，程序员还应该注意，由于为了避免和 C++98 中 auto 的含义发生混淆，C++11 只保

留 auto 作为类型指示符的用法，以下的语句在 C++98 和 C 语言中都是合法的，但在 C++11 中，编译器则会报错。

```
auto int i = 1;
```

总的来说，auto 在 C++11 中是相当关键的特性之一。我们之后还会在很多地方看到 auto，比如 4.4 节中的追踪返回类型的函数声明，以及 7.3 节中 lambda 与 auto 的配合使用等。（事实上，第 3 章中我们也使用过）。不过，如我们提到的，auto 只是 C++11 中类型推导体现的一部分。其余的，则会在 decltype 中得到体现。

4.3 decltype

☞ **类别：库作者**

4.3.1 typeid 与 decltype

我们在 4.2 节中曾经提到过静态类型和动态类型的区别。不过与 C 完全不支持动态类型不同的是，C++ 在 C++98 标准中就部分支持动态类型了。如读者能够想象的，C++98 对动态类型支持就是 C++ 中的运行时类型识别（RTTI）。

RTTI 的机制是为每个类型产生一个 type_info 类型的数据，程序员可以在程序中使用 typeid 随时查询一个变量的类型，typeid 就会返回变量相应的 type_info 数据。而 type_info 的 name 成员函数可以返回类型的名字。而在 C++11 中，又增加了 hash_code 这个成员函数，返回该类型唯一的哈希值，以供程序员对变量的类型随时进行比较。我们可以看看代码清单 4-16 所示的例子。

代码清单 4-16

```
#include <iostream>
#include <typeinfo>
using namespace std;

class White{};
class Black{};

int main() {
    White a;
    Black b;

    cout << typeid(a).name() << endl;    // 5White
    cout << typeid(b).name() << endl;    // 5Black

    White c;

    bool a_b_sametype = (typeid(a).hash_code() == typeid(b).hash_code());
```

```
    bool a_c_sametype = (typeid(a).hash_code() == typeid(c).hash_code());

    cout << "Same type? " << endl;
    cout << "A and B? " << (int)a_b_sametype << endl;    // 0
    cout << "A and C? " << (int)a_c_sametype << endl;    // 1
}

// 编译选项 :g++ -std=c++11 4-3-1.cpp
```

这里我们定义了两个不同的类型 White 和 Black，以及其类型的变量 a 和 b。此外我们使用 typeid 返回类型的 type_info，并分别应用 name 打印类型的名字（5 这样的前缀是 g++ 这类编译器输出的名字，其他编译器可能会打印出其他的名字，这个标准并没有明确规定），应用 hash_code 进行类型的比较。在 RTTI 的支持下，程序员可以在一定程度上了解程序中类型的信息（这里可以注意一下，相比于 4.1.2 节中的 is_same 模板函数的成员类型 value 在编译时得到信息，hash_code 是运行时得到的信息）。

除了 typeid 外，RTTI 还包括了 C++ 中的 dynamic_cast 等特性。不过不得不提的是，由于 RTTI 会带来一些运行时的开销，所以一些编译器会让用户选择性地关闭该特性（比如 XL C/C++ 编译器的 -qnortti，GCC 的选项 -fno-rttion，或者微软编译器选项 /GR-）。而且很多时候，运行时才确定出类型对于程序员来说为时过晚，程序员更多需要的是在编译时期确定出类型（标准库中非常常见）。而通常程序员是要使用这样的类型而不是识别该类型，因此 RTTI 无法满足需求。

事实上，在 C++ 的发展中，类型推导是随着模板和泛型编程的广泛使用而引入的。在非泛型的编程中，我们不用对类型进行推导，因为任何表达式中变量的类型都是明确的，而运算、函数调用等也都有明确的返回类型。然而在泛型的编程中，类型成了未知数。我们可以回顾一下 4.2 节中代码清单 4-9 所示的例子，其中的模板函数 Sum 的参数的 t1 和 t2 类型都是不确定的，因此 t1 + t2 这个表达式将返回的类型也就不可由 Sum 的编写者确定。无疑，这样的状况会限制模板的使用范围和编写方式。而最好的解决办法就是让编译器辅助地进行类型推导。

在 decltype 产生之前，很多编译器的厂商都开发了自己的 C++ 语言扩展用于类型推导。比如 GCC 的 typeof 操作符就是其中的一种。C++11 则将这些类型推导手段进行了细致的考量，最终标准化为 auto 以及 decltype。与 auto 类似地，decltype 也能进行类型推导，不过两者的使用方式却有一定的区别。我们可以看代码清单 4-17 所示的这个简单的例子。

代码清单 4-17

```
#include <typeinfo>
#include <iostream>
using namespace std;
```

```
int main() {
    int i;
    decltype(i) j = 0;
    cout << typeid(j).name() << endl;    // 打印出 "i", g++ 表示 int

    float a;
    double b;
    decltype(a + b) c;
    cout << typeid(c).name() << endl;    // 打印出 "d", g++ 表示 double
}

// 编译选项 :g++ -std=c++11 4-3-2.cpp
```

在代码清单 4-17 中，我们看到变量 j 的类型由 decltype(i) 进行声明，表示 j 的类型跟 i 相同（或者准确地说，跟 i 这个表达式返回的类型相同）。而 c 的类型则跟 (a + b) 这个表达式返回的类型相同。而由于 a + b 加法表达式返回的类型为 double（a 会被扩展为 double 类型与 b 相加），所以 c 的类型被 decltype 推导为 double。

从这个例子中可以看到，decltype 的类型推导并不是像 auto 一样是从变量声明的初始化表达式获得变量的类型，decltype 总是以一个普通的表达式为参数，返回该表达式的类型。而与 auto 相同的是，作为一个类型指示符，decltype 可以将获得的类型来定义另外一个变量。与 auto 相同，decltype 类型推导也是在编译时进行的。

4.3.2 decltype 的应用

在 C++11 中，使用 decltype 推导类型是非常常见的事情。比较典型的就是 decltype 与 typdef/using 的合用。在 C++11 的头文件中，我们常能看以下这样的代码：

```
using size_t = decltype(sizeof(0));
using ptrdiff_t = decltype((int*)0 - (int*)0);
using nullptr_t = decltype(nullptr);
```

这里 size_t 以及 ptrdiff_t 还有 nullptr_t（参见 7.1 节）都是由 decltype 推导出的类型。这种定义方式非常有意思。在一些常量、基本类型、运算符、操作符等都已经被定义好的情况下，类型可以按照规则被推导出。而使用 using，就可以为这些类型取名。这就颠覆了之前类型拓展需要将扩展类型“映射”到基本类型的常规做法。

除此之外，decltype 在某些场景下，可以极大地增加代码的可读性。比如代码清单 4-18 所示的例子。

代码清单 4-18

```
#include <vector>
using namespace std;
```

```
int main() {
    vector<int> vec;
    typedef decltype(vec.begin()) vectype;

    for (vectype i = vec.begin(); i < vec.end(); i++) {
        // 做一些事情
    }
    for (decltype(vec)::iterator i = vec.begin(); i < vec.end(); i++) {
        // 做一些事情
    }
}

// 编译选项 :g++ -std=c++11 4-3-3.cpp
```

在代码清单 4-18 中，我们定义了 vector 的 iterator 的类型。这个类型还可以在 main 函数中重用。当我们遇到一些具有复杂类型的变量或表达式时，就可以利用 decltype 和 typedef/using 的组合来将其转化为一个简单的表达式，这样在以后的代码写作中可以提高可读性和可维护性。此外我们可以看到 decltype(vec)::iterator 这样的灵活用法，这看起来跟 auto 非常类似，也类似于是一种“占位符”式的替代。

在 C++ 中，我们有时会遇到匿名的类型，而拥有了 decltype 这个利器之后，重用匿名类型也并非难事。我们可以看看代码清单 4-19 所示的例子。

代码清单 4-19

```
enum class{K1, K2, K3}anon_e;    // 匿名的强类型枚举

union {
    decltype(anon_e) key;
    char* name;
}anon_u;     // 匿名的 union 联合体

struct {
    int d;
    decltype(anon_u) id;
}anon_s[100];    // 匿名的 struct 数组

int main() {
    decltype(anon_s) as;
    as[0].id.key = decltype(anon_e)::K1;    // 引用匿名强类型枚举中的值
}

// 编译选项 :g++ -std=c++11 4-3-4.cpp
```

这里我们使用了 3 种不同的匿名类型：匿名的强类型枚举 anon_e（请参见 5.1 节）、匿名的联合体 anon_u，以及匿名的结构体数组 anon_s。可以看到，只要通过匿名类型的变量名 anon_e、anon_u，以及 anon_s，decltype 可以推导其类型并且进行重用。这些都是以前 C++

代码所做不到的。事实上，在一些C代码中，匿名的结构体和联合体并不少见。不过匿名一般都有匿名理由，一般程序员都不希望匿名后的类型被重用。这里的decltype只是提供了一种语法上的可能。

进一步地，有了decltype，我们可以适当扩大模板泛型的能力。还是以代码清单4-9为例，如果我们稍微改变一下函数模板的接口，该模板将适用于更大的范围。我们来看看代码清单4-20中经过改进的例子。

代码清单4-20

```
// s的类型被声明为decltype(t1 + t2)
template<typename T1, typename T2>
void Sum(T1 & t1, T2 & t2, decltype(t1 + t2) & s) {
    s = t1 + t2;
}

int main() {
    int a = 3;
    long b = 5;
    float c = 1.0f, d = 2.3f;

    long e;
    float f;
    Sum(a, b, e);    // s的类型被推导为long
    Sum(c, d, f);    // s的类型被推导为float
}

// 编译选项 :g++ -std=c++11 4-3-5.cpp
```

相比于代码清单4-9的例子，代码清单4-20的Sum函数模板增加了类型为decltype(t1+t2)的s作为参数，而函数本身不返回任何值。这样一来，Sum的适用范围增加，因为其返回的类型不再是代码清单4-9中单一的double类型，而是根据t1 + t2推导而来的类型。不过这里还是有一定的限制，我们可以看到返回值的类型必须一开始就被指定，程序员必须清楚Sum运算的结果使用什么样的类型来存储是合适的，这在一些泛型编程中依然不能满足要求。解决的方法是结合decltype与auto关键字，使用追踪返回类型的函数定义来使得编译器对函数返回值进行推导。我们会在4.4节中看到具体的细节（事实上，decltype一个最大的用途就是用在追踪返回类型的函数中）。

在代码清单4-20中模板定义虽然存在一些限制，但也基本是可以广泛使用的。但是不得不提的是，某些情况下，模板库的使用人员可能认为一些自然而简单的数据结构，比如数组，也是可以被模板类所包括的。不过很明显，如果t1和t2是两个数组，t1 + t2不会是合法的表达式。为了避免不必要的误解，模板库的开发人员应该为这些特殊的情况提供其他的版本，如代码清单4-21所示。

代码清单 4-21

```
template<typename T1, typename T2>
void Sum(T1 & t1, T2 & t2, decltype(t1 + t2) & s) {
    s = t1 + t2;
}

void Sum(int a[], int b[], int c[]){
    // 数组版本
}

int main() {
    int a[5], b[5], c[5];
    Sum(a, b, c);    // 选择数组版本

    int d, e, f;
    Sum(d, e, f);    // 选择模板的实例化版本
}

// 编译选项 :g++ -std=c++11 4-3-6.cpp
```

在代码清单 4-21 中，由于声明了数组版本 Sum，编译器在编译 Sum(a, b, c) 的时候，会优先选择数组版本，而编译 Sum(d, e, f) 的时候，依然会对应到模板的实例化版本。这就能够保证 Sum 模板函数最大的可用性（不过这里的数组版本似乎做不了什么事情，因为数组长度丢失了）。

我们在实例化一些模板的时候，decltype 也可以起到一些作用，我们可以看看代码清单 4-22 所示的例子。

代码清单 4-22

```
#include <map>
using namespace std;

int hash(char*);

map<char*, decltype(hash)> dict_key;     // 无法通过编译
map<char*, decltype(hash(nullptr))> dict_key1;

// 编译选项 :g++ -c -std=c++11 4-3-7.cpp
```

在代码清单 4-22 中，我们实例化了标准库中的 map 模板。因为该 map 是为了存储字符串以及与其对应哈希值的，因此我们可以通过 decltype(hash(nullptr)) 来确定哈希值的类型。这样的定义非常直观，但是程序员必须要注意的是，decltype 只能接受表达式做参数，像函数名做参数的表达式 decltype(hash) 是无法通过编译的。

事实上，decltype 在 C++11 的标准库中也有一些应用，一些标准库的实现也会依赖于

decltype 的类型推导。一个典型的例子是基于 decltype 的模板类 result_of，其作用是推导函数的返回类型。我们可以看一下应用的实例，如代码清单 4-23 所示。

代码清单 4-23

```
#include <type_traits>
using namespace std;

typedef double (*func)();

int main() {
    result_of<func()>::type f;    // 由 func() 推导其结果类型
}

// 编译选项 :g++ -std=c++11 4-3-8.cpp
```

这里 f 的类型最终被推导为 double，而 result_of 并没有真正调用 func() 这个函数，这一切都是因为底层的实现使用了 decltype。result_of 的一个可能的实现方式如下：

```
template<class>
struct result_of;

template<class F, class... ArgTypes>
struct result_of<F(ArgTypes...)>
{
    typedef decltype(
                std::declval<F>()(std::declval<ArgTypes>()...)
                ) type;
};
```

请读者忽略 declval [⊖]，这里标准库将 decltype 作用于函数调用上（使用了变长函数模板），并将函数调用表达式返回的类型 typedef 为一个名为 type 的类型。这样一来，代码清单 4-23 中的 result_of<func()>::type 就会被 decltype 推导为 double。

4.3.3 decltype 推导四规则

作为 auto 的伙伴，decltype 在 C++11 中也非常重要。不过跟 auto 一样，由于应用广泛，所以使用 decltype 也有很多的细则条款需要注意。很多时候，用户会发现 decltype 的行为并不如预期，那么下面的规则可能会更好地解释这些“不如预期”的编译器行为。

大多数时候，decltype 的使用看起来非常平易近人，但是有时我们也会落入一些令人疑惑的陷阱。最典型的就是代码清单 4-24 所示的这个例子。

⊖ 实际是 STL 中的一种语法技巧，更多的内容可以查阅一些在线文档，如 http://en.cppreference.com/w/cpp/utility/declval。

代码清单 4-24

```
int i;
decltype(i) a;        // a: int
decltype((i)) b;      // b: int &, 无法编译通过

// 编译选项 :g++ -std=c++11 4-3-9.cpp
```

我们在编译代码清单 4-24 的时候，会惊奇地发现，decltype((i)) b; 这样的语句编译不过。编译器会提示 b 是一个引用，但没有被赋初值。而 decltype(i) a; 这一句却能通过编译，因为其类型被如预期地推导为 int。

这种问题显得非常诡异，单单多了一对圆括号，decltype 所推导出的类型居然发生了变化。事实上，C++11 中 decltype 推导返回类型的规则比我们想象的复杂。具体地，当程序员用 decltype(e) 来获取类型时，编译器将依序判断以下四规则：

1）如果 e 是一个没有带括号的标记符表达式（id-expression）或者类成员访问表达式，那么 decltype(e) 就是 e 所命名的实体的类型。此外，如果 e 是一个被重载的函数，则会导致编译时错误。

2）否则，假设 e 的类型是 T，如果 e 是一个将亡值 (xvalue)，那么 decltype(e) 为 T&&。

3）否则，假设 e 的类型是 T，如果 e 是一个左值，则 decltype(e) 为 T&。

4）否则，假设 e 的类型是 T，则 decltype(e) 为 T。

这里我们要解释一下标记符表达式（id-expression）。基本上，所有除去关键字、字面量等编译器需要使用的标记之外的程序员自定义的标记（token）都可以是标记符（identifier）。而单个标记符对应的表达式就是标记符表达式。比如程序员定义了：

```
int arr[4];
```

那么 arr 是一个标记符表达式，而 arr[3] + 0, arr[3] 等，则都不是标记符表达式。

我们再回到代码清单 4-24，并结合 decltype 类型推导的规则，就可以知道，decltype(i) a; 使用了推导规则 1——因为 i 是一个标记符表达式，所以类型被推导为 int。而 decltype((i)) b; 中，由于 (i) 不是一个标记符表达式，但却是一个左值表达式（可以有具名的地址），因此，按照 decltype 推导规则 3，其类型应该是一个 int 的引用。

上面的规则看起来非常复杂，但事实上，在实际应用中，decltype 类型推导规则中最容易引起迷惑的只有规则 1 和规则 3。我们可以通过代码清单 4-25 所示的这个例子再加深一下理解。

代码清单 4-25

```
int i = 4;
```

```
int arr[5] = {0};
int *ptr = arr;

struct S { double d; } s;

void Overloaded(int);
void Overloaded(char);        // 重载的函数

int && RvalRef();

const bool Func(int);

// 规则 1：单个标记符表达式以及访问类成员，推导为本类型
decltype(arr) var1;              // int[5]，标记符表达式
decltype(ptr) var2;              // int*，标记符表达式
decltype(s.d) var4;              // double，成员访问表达式
decltype(Overloaded) var5;       // 无法通过编译，是个重载的函数

// 规则 2：将亡值，推导为类型的右值引用
decltype(RvalRef()) var6 = 1;   // int&&

// 规则 3：左值，推导为类型的引用
decltype(true ? i : i) var7 = i;     // int&，三元运算符，这里返回一个 i 的左值
decltype((i)) var8 = i;              // int&，带圆括号的左值
decltype(++i) var9 = i;              // int&，++i 返回 i 的左值
decltype(arr[3]) var10 = i;          // int& [] 操作返回左值
decltype(*ptr)  var11 = i;           // int& * 操作返回左值
decltype("lval") var12 = "lval";     // const char(&)[9]，字符串字面常量为左值

// 规则 4：以上都不是，推导为本类型
decltype(1) var13;                 // int，除字符串外字面常量为右值
decltype(i++) var14;               // int，i++ 返回右值
decltype((Func(1))) var15;         // const bool，圆括号可以忽略

// 编译选项 :g++ -std=c++11 -c 4-3-10.cpp
```

代码清单 4-25 中我们将四种规则的例子都列了出来。可以看到，规则 1 不但适用于基本数据类型，还适用于指针、数组、结构体，甚至函数类型的推导，事实上，规则 1 在 decltyp 类型推导中运用的最为广泛。而规则 2 则比较简单，基本上符合程序员的想象。

规则 3 其实是一个左值规则。decltype 的参数不是标志表达式或者类成员访问表达式，且参数都为左值，推导出的类型均为左值引用。规则 4 则是适用于以上都不适用者。我们这里看到了 ++i 和 i++ 在左右值上的区别，以及字符串字面常量 lval、非字符串字面常量 1 在左右值间的区别。

看过这么多规则，读者可能觉得过于复杂，但事实上，如同我们之前提到的，引起麻烦的只是规则 3 带来的左值引用的推导。一个简单的能够让编译器提示的方法是，如果使用

decltype 定义变量，那么先声明这个变量，再在其他语句里对其进行初始化。这样一来，由于左值引用总是需要初始化的，编译器会报错提示。另外一些时候，C++11 标准库中添加的模板类 is_lvalue_reference，可以帮助程序员进行一些推导结果的识别。我们看看代码清单 4-26 所示的例子。

代码清单 4-26

```
#include <type_traits>
#include <iostream>
using namespace std;

int i = 4;
int arr[5] = {0};
int *ptr = arr;

int && RvalRef();

int main(){
    cout << is_rvalue_reference<decltype(RvalRef())>::value << endl;     // 1

    cout << is_lvalue_reference<decltype(true ? i : i)>::value << endl; // 1
    cout << is_lvalue_reference<decltype((i))>::value << endl;          // 1
    cout << is_lvalue_reference<decltype(++i)>::value << endl;          // 1
    cout << is_lvalue_reference<decltype(arr[3])>::value << endl;       // 1
    cout << is_lvalue_reference<decltype(*ptr)>::value << endl;         // 1
    cout << is_lvalue_reference<decltype("lval")>::value << endl;       // 1

    cout << is_lvalue_reference<decltype(i++)>::value << endl;          // 0
    cout << is_rvalue_reference<decltype(i++)>::value << endl;          // 0
}

// 编译选项 :g++ -std=c++11 4-3-11.cpp
```

代码清单 4-26 中，我们使用了模板类 is_lvalue_reference 的成员 value 来查看 decltype 的效果（1 表示是左值引用，0 则反之）。如我们所见，代码清单 4-26 中凡是符合规则 3 的，都会被推导为左值引用。如果程序员在程序的书写中不是非常确定 decltype 是否将类型推导为左值引用，也可以通过这样的小实验来辅助确定。这里我们还使用了模板函数 is_rvalue_reference，同样，程序员也可以通过它来确定 decltype 是否推导出了右值引用。

4.3.4 cv 限制符的继承与冗余的符号

与 auto 类型推导时不能“带走”cv 限制符不同，decltype 是能够“带走”表达式的 cv 限制符的。不过，如果对象的定义中有 const 或 volatile 限制符，使用 decltype 进行推导时，其成员不会继承 const 或 volatile 限制符。我们可以看看如代码清单 4-27 所示的例子。

代码清单 4-27

```
#include <type_traits>
#include <iostream>
using namespace std;

const int ic = 0;
volatile int iv;

struct S { int i; };

const S a = {0};
volatile S b;
volatile S* p = &b;

int main() {
    cout << is_const<decltype(ic)>::value << endl;       // 1
    cout << is_volatile<decltype(iv)>::value << endl;    // 1

    cout << is_const<decltype(a)>::value << endl;        // 1
    cout << is_volatile<decltype(b)>::value << endl;     // 1

    cout << is_const<decltype(a.i)>::value << endl;      // 0, 成员不是const
    cout << is_volatile<decltype(p->i)>::value << endl; // 0, 成员不是volatile
}

// 编译选项:g++ -std=c++11 4-3-12.cpp
```

代码清单 4-27 的例子中，我们使用了 C++ 库提供的 is_const 和 is_volatile 来查看类型是否是常量或者易失的。可以看到，结构体变量 a、b 和结构体指针 p 的 cv 限制符并没有出现在其成员的 decltype 类型推导结果中。

而与 auto 相同的，decltype 从表达式推导出类型后，进行类型定义时，也会允许一些冗余的符号。比如 cv 限制符以及引用符号 &，通常情况下，如果推导出的类型已经有了这些属性，冗余的符号则会被忽略，如代码清单 4-28 所示。

代码清单 4-28

```
#include <type_traits>
#include <iostream>
using namespace std;

int i = 1;
int & j = i;
int * p = &i;
const int k = 1;

int main() {
    decltype(i) & var1 = i;
```

```
    decltype(j) & var2 = i;      // 冗余的 &，被忽略

    cout << is_lvalue_reference<decltype(var1)>::value << endl;// 1，是左值引用

    cout << is_rvalue_reference<decltype(var2)>::value << endl;// 0，不是右值引用
    cout << is_lvalue_reference<decltype(var2)>::value << endl;// 1，是左值引用

    decltype(p)* var3 = &i;      // 无法通过编译
    decltype(p)* var3 = &p;      // var3 的类型是 int**

    auto* v3 = p;                // v3 的类型是 int*
    v3 = &i;

    const decltype(k) var4 = 1; // 冗余的 const，被忽略
}

// 编译选项 :g++ -std=c++11 4-3-13.cpp
```

在代码清单 4-28 中，我们定义了类型为 decltype(i) & 的变量 var1，以及类型为 decltype(j) & 的变量 var2。由于 i 的类型为 int，所以这里的引用符号保证 var1 成为一个 int& 引用类型。而由于 j 本来就是一个 int & 的引用类型，所以 decltype 之后的 & 成为了冗余符号，会被编译器忽略，因此 j 的类型依然是 int &。

这里特别要注意的是 decltype(p)* 的情况。可以看到，在定义 var3 变量的时候，由于 p 的类型是 int*，因此 var3 被定义为了 int** 类型。这跟 auto 声明中，* 也可以是冗余的不同。在 decltype 后的 * 号，并不会被编译器忽略。

此外我们也可以看到，var4 中 const 可以被冗余的声明，但会被编译器忽略，同样的情况也会发生在 volatile 限制符上。

总的说来，decltype 算得上是 C++11 中类型推导使用方式上最灵活的一种。虽然看起来它的推导规则比较复杂，有的时候跟 auto 推导结果还略有不同，但大多数时候，我们发现使用 decltype 还是自然而亲切的。一些细则的区别，读者可以在使用时遇到问题再返回查验。而下面的追踪返回类型的函数定义，则将融合 auto、decltype，将 C++11 中的泛型能力提升到更高的水平。

4.4 追踪返回类型

☞ **类别：库作者**

4.4.1 追踪返回类型的引入

如我们在 4.2 节与 4.3 节中反复提到的，追踪返回类型配合 auto 与 decltype 会真正释放 C++11 中泛型编程的能力。

在C++98中，如果一个函数模板的返回类型依赖于实际的入口参数类型，那么该返回类型在模板实例化之前可能都无法确定，这样的话我们在定义该函数模板时就会遇到麻烦。可以回想一下代码清单4-9的例子，由于Sum模板函数的两个参数t1与t2的类型没有确定，所以我们只能简单地设置结果s为double类型并返回。这就限制了Sum的使用范围（大概只能用于数值不算太大的算术运算）。而在代码清单4-20中，我们改进了Sum模板函数，通过增加decltype(t1+t2)的参数的方式来返回泛型的值。这样做虽然扩大了Sum的适用范围，但改变了Sum的使用方式，在一些情况下，也是不可以接受的。而且由于程序员必须预先知道返回的类型，其使用上的灵活性也就打了一些折扣。

那么，最为直观的解决方式就是对返回类型进行类型推导。而最为直观的书写方式如下所示：

```
template<typename T1, typename T2>
decltype(t1 + t2) Sum(T1 & t1, T2 & t2) {
    return t1 + t2;
}
```

这样的写法虽然看似不错，不过对编译器来说有些小问题。编译器在推导decltype(t1 + t2)时的，表达式中的t1和t2都未声明（虽然它们近在咫尺，编译器却只会从左往右地读入符号）。按照C/C++编译器的规则，变量使用前必须已经声明，因此，为了解决这个问题，C++11引入新语法——追踪返回类型，来声明和定义这样的函数。

```
template<typename T1, typename T2>
auto Sum(T1 & t1, T2 & t2) -> decltype(t1 + t2){
    return t1 + t2;
}
```

如上面的写法所示，我们把函数的返回值移至参数声明之后，复合符号-> decltype(t1 + t2)被称为追踪返回类型。而原本函数返回值的位置由auto关键字占据。这样，我们就可以让编译器来推导Sum函数模板的返回类型了。而auto占位符和->return_type也就是构成追踪返回类型函数的两个基本元素。

4.4.2 使用追踪返回类型的函数

追踪返回类型的函数和普通函数的声明最大的区别在于返回类型的后置。在一般情况下，普通函数的声明方式会明显简单于最终返回类型。比如：

```
int func(char* a, int b);
```

这样的书写会比下面的书写少上不少。

```
auto func(char*a, int b) -> int;
```

不过有的时候，追踪返回类型声明的函数也会带给大家一些意外，比如代码清单4-29所

示的这个例子。

代码清单 4-29

```
class OuterType{
    struct InnerType { int i; };

    InnerType GetInner();
    InnerType it;
};

// 可以不写 OuterType::InnerType
auto OuterType::GetInner() -> InnerType {
    return it;
}

// 编译选项 :g++ -std=c++11 4-4-1.cpp
```

在代码清单 4-29 中，可以看到我们使用最终返回类型的时候，InnerType 不必写明其作用域。这对于讨厌写很长作用域的程序员来说，也算得上是一个好消息。

如我们刚才提到的，返回类型后置，使模板中的一些类型推导就成为了可能。我们可以看看代码清单 4-30 所示的使用追踪返回类型的例子。

代码清单 4-30

```
#include <iostream>
using namespace std;

template<typename T1, typename T2>
auto Sum(const T1 & t1, const T2 & t2) -> decltype(t1 + t2){
    return t1 + t2;
}

template <typename T1, typename T2>
auto Mul(const T1 & t1, const T2 & t2) -> decltype(t1 * t2){
    return t1 * t2;
}

int main() {
    auto a = 3;
    auto b = 4L;
    auto pi = 3.14;

    auto c = Mul(Sum(a, b), pi);
    cout << c << endl;  // 21.98
}

// 编译选项 :g++ -std=c++11 4-4-2.cpp
```

在代码清单 4-30 的例子中，我们定义了两个模板函数 Sum 和 Mul，它们的参数的类型和返回值都在实例化时决定。而由于 main 函数中还使用了 auto，整个例子中没有看到一个“具体”的类型声明。事实上，这段代码尤其是主函数，看起来有点像是一个动态类型语言的代码，而不像是一个有着严格静态类型的 C++ 的代码。当然，这一切都要归功于类型推导帮助下的泛型编程。程序员在编写代码时无需关心任何时段的类型选择，编译器会合理地进行推导，而简单程序的书写也由此得到了极大的简化。

除了解决以上所描述的问题，追踪返回类型的另一个优势是简化函数的定义，提高代码的可读性。这种情况常见于函数指针中。我们可以看一下代码清单 4-31 所示的例子。

代码清单 4-31

```
#include <type_traits>
#include <iostream>
using namespace std;

// 有的时候，你会发现这是面试题
int (*(*pf())())() {
    return nullptr;
}

// auto (*)() -> int(*) () 一个返回函数指针的函数（假设为 a 函数）
// auto pf1() -> auto (*)() -> int (*)() 一个返回 a 函数的指针的函数
auto pf1() -> auto (*)() -> int (*)() {
    return nullptr;
}

int main() {
    cout << is_same<decltype(pf), decltype(pf1)>::value << endl;    // 1
}

// 编译选项 :g++ -std=c++11 4-4-3.cpp
```

在代码清单 4-31 中，定义了两个类型完全一样的函数 pf 和 pf1。其返回的都是一个函数指针。而该函数指针又指向一个返回函数指针的函数。这一点通过 is_same 的成员 value 已经能够确定了（参见 4.1.1）。而仔细看一看函数类型的声明，可以发现老式的声明法可读性非常差。而追踪返回类型只需要依照从右向左的方式，就可以将嵌套的声明解析出来。这大大提高了嵌套函数这类代码的可读性。

除此之外，追踪返回类型也被广泛地应用在转发函数中，如代码清单 4-32 所示。

代码清单 4-32

```
#include <iostream>
using namespace std;
```

```
double foo(int a) {
    return (double)a + 0.1;
}

int foo(double b) {
    return (int)b;
}

template <class T>
auto Forward(T t) -> decltype(foo(t)){
    return foo(t);
}

int main(){
    cout << Forward(2) << endl;      // 2.1
    cout << Forward(0.5) << endl;    // 0
}

// 编译选项 :g++ -std=c++11 4-4-4.cpp
```

代码清单 4-32 中，我们可以看到，由于使用了追踪返回类型，可以实现参数和返回类型不同时的转发。

追踪返回类型还可以用在函数指针中，其声明方式与追踪返回类型的函数比起来，并没有太大的区别。比如：

```
auto (*fp)() -> int;
```

和

```
int (*fp)();
```

的函数指针声明是等价的。同样的情况也适用于函数引用，比如：

```
auto (&fr)() -> int;
```

和

```
int (&fr)();
```

的声明也是等价的。

除了以上所描述的函数模板、普通函数、函数指针、函数引用以外，追踪返回类型还可以用在结构或类的成员函数、类模板的成员函数里，其方法大同小异，这里不一一举例了。另外，没有返回值的函数也可以被声明为追踪返回类型，程序员只需要将返回类型声明为 void 即可。

4.5 基于范围的 for 循环

☞ **类别：所有人**

在 C++98 标准中，如果要遍历一个数组，通常会需要代码清单 4-33 所示的代码。

代码清单 4-33

```
#include <iostream>
using namespace std;

int main() {
    int arr[5] = { 1, 2, 3, 4, 5};
    int * p;
    for (p = arr; p < arr + sizeof(arr)/sizeof(arr[0]); ++p){
        *p *= 2;
    }
    for (p = arr; p < arr + sizeof(arr)/sizeof(arr[0]); ++p){
        cout << *p << '\t';
    }
}

// 编译选项 :g++ 4-5-1.cpp
```

代码清单 4-33 中，我们使用了指针 p 来遍历数组 arr 中的内容，两个循环分别完成了每个元素自乘以 2 和打印工作。而 C++ 的标准模板库中，我们还可以找到形如 for_each 的模板函数。如果我们使用 for_each 来完成代码清单 4-33 中的工作，代码看起来会是代码清单 4-34 所示的样子。

代码清单 4-34

```
#include <algorithm>
#include <iostream>
using namespace std;

int action1(int & e){ e *= 2; }
int action2(int & e){ cout << e << '\t'; }

int main() {
    int arr[5] = { 1, 2, 3, 4, 5};
    for_each(arr, arr + sizeof(arr)/sizeof(arr[0]), action1);
    for_each(arr, arr + sizeof(arr)/sizeof(arr[0]), action2);
}

// 编译选项 :g++ -std=c++11 -c
```

for_each 使用了迭代器的概念，其迭代器就是指针。由于迭代器内含了自增操作的概

念，所以如代码清单 4-33 中的 ++p 操作则可以不写在 for_each 循环中了。不过无论是代码清单 4-33 还是代码清单 4-34，都需要告诉循环体其界限的范围，即 arr 到 arr + sizeof(arr)/sizeof(arr[0]) 之间，才能按元素执行操作。

事实上，循环的“自动范围”这个问题，在很多语言中已经实现了。我们可以看看 bash 中 for 循环的使用方法。

```
for i in '1 2 3 4 5' ;
do
    $i = `expr $i + $i`;
echo $i;
done
```

上面的 bash 完成了与代码清单 4-33 及代码清单 4-34 一样的功能，不过语法上，bash 使用了 for ... in 的方式，因此循环的范围是“自说明”的，是在‘1 2 3 4 5’这样的范围中完成元素操作的。很多时候，对于一个有范围的集合而言，由程序员来说明循环的范围是多余的，也是容易犯错误的。而 C++11 也引入了基于范围的 for 循环，就可以很好地解决了这个问题。

我们可以看一下基于范围的 for 循环改写的例子，如代码清单 4-35 所示。

代码清单 4-35

```
#include <iostream>
using namespace std;

int main() {
    int arr[5] = { 1, 2, 3, 4, 5 };
    for (int & e: arr)
        e *= 2;

    for (int & e: arr)
        cout << e << '\t';
}

// 编译选项 :g++ -std=c++11 4-5-3.cpp
```

代码清单 4-35 就是一个基于范围的 for 循环的实例。for 循环后的括号由冒号“：”分为两部分，第一部分是范围内用于迭代的变量，第二部分则表示将被迭代的范围。在代码清单 4-35 这个具体的例子当中，表示的是在数组 arr 中用迭代器 e 进行遍历。这样一来，遍历数组和 STL 容器就非常容易了。

在代码清单 4-35 中，基于范围的 for 循环中迭代的变量采用了引用的形式，如果迭代变量的值在循环中不会被修改，那我们完全可以不用引用的方式来做迭代变量。比如上例中的第二个基于范围的 for 循环可以被改为：

```
for (int e: arr)
    cout << e << '\t';
```

代码依然可以很好地工作。当然，如果结合之前讲过的 auto 类型指示符，循环会显得更简练。

```
for (auto e: arr)
    cout << e << '\t';
```

基于范围的 for 循环跟普通循环是一样的，可以用 continue 语句来跳过循环的本次迭代，而用 break 语句来跳出整个循环。

值得指出的是，是否能够使用基于范围的 for 循环，必须依赖于一些条件。首先，就是 for 循环迭代的范围是可确定的。对于类来说，如果该类有 begin 和 end 函数，那么 begin 和 end 之间就是 for 循环迭代的范围。对于数组而言，就是数组的第一个和最后一个元素间的范围。其次，基于范围的 for 循环还要求迭代的对象实现 ++ 和 == 等操作符。对于标准库中的容器，如 string、array、vector、deque、list、queue、map、set 等，不会有问题，因为标准库总是保证其容器定义了相关的操作。普通的已知长度的数组也不会有问题。而用户自己写的类，则需要自行提供相关操作。

相反，如果我们数组大小不能确定的话，是不能够使用基于范围的 for 循环的，比如代码清单 4-36 所示的用法，就会导致编译时的错误。

代码清单 4-36

```
#include <iostream>
using namespace std;

int func(int a[]) {

    for (auto e: a)
        cout << e;
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    func(arr);
}

// 编译选项 :g++ -std=c++11 4-5-4.cpp
```

上述代码会报错，因为作为参数传递而来的数组 a 的范围不能确定，因此也就不能使用基于范围循环 for 循环对其进行迭代的操作。

另外一点，习惯了使用迭代器的 C++ 程序员可能需要注意，就是基于范围的循环使用在标准库的容器中时，如果使用 auto 来声明迭代的对象的话，那么这个对象不会是迭代器对

象。代码清单 4-37 所示的这个简单的例子可以说明这一情况。

代码清单 4-37

```
#include <vector>
#include <iostream>
using namespace std;

int main() {
    vector<int> v = {1, 2, 3, 4, 5};
    for (auto i = v.begin(); i != v.end(); ++i)
        cout << *i << endl;      // i 是迭代器对象

    for (auto e: v)
        cout << e << endl;       // e 是解引用后的对象
}

// 编译选项 :g++ -std=c++11 4-5-5.cpp
```

读者只需要注意 e 和 *i 的区别就可以了。

4.6 本章小结

在本章里，介绍了 C++11 四个讨人喜欢的新特性，它们的特色非常鲜明，都能够减少代码的书写，或加强代码的可读性。

首先，我们看到 C++11 中解决了双右尖括号的语法问题的小改进。相比于 C++98 中，模板实例化时右尖括号间必须空格的“奇怪”规定，C++11 可以说采取了更加平易近人的态度，使得这一规则不再需要。

其次，我们可以看到 C++11 中关于类型推导的巨大改进。虽然 auto、decltype，以及追踪返回类型的函数声明，它们的由来都可以追溯到 C++ 使用模板进行泛型编程上，但从实际效果上看，由于有了类型推导，整个 C++ 程序的书写的便利性被极大地提高了。相应地，代码可读性也大大改善。可以说，类型推导不仅提高了模板库的泛型能力，导致 C++11 风格下的编程跟以前的 C++98 风格下的编程有了改变，而且在我们的范例中看到了全部倚仗于类型推导，没有一个“明确”类型的 C++ 代码，这对于一个静态类型的、有着长久历史渊源的语言而言，几乎是不可想象的。但是在 C++11 中，这种友好的编程方式已经得到了良好的支持。虽然深入语言细节的时候，我们可能发现一些推导的规则依然复杂，但对于 90% 以上的普通应用，类型推导已经做得足够好用。如果要在 C++11 中挑选最好的新特性的话，类型推导无疑会是非常有力的竞争者。

再者，我们看到了基于范围的 for 循环。结合 auto 关键字，程序员只需要知道“我在迭代地访问每个元素”即可，而再也不必关心范围、如何迭代访问等细节。这比以前标准库的

for_each 做得更加出色。虽然基本上基于范围的 for 循环没有任何灵活性可言，但将常做的事情做得更快更好，也往往是程序员最大的需求。

总的来说，以上新特性对于新手来说，非常易学，对于老兵而言，非常好用。无论对什么水平的编程者来说，总可以从使用这些特性当中获得一些益处。在后面的章节里，我们还会继续看到这些特性的身影（就如在前面的章节里一样）。

第⑤章

提高类型安全

相比于 C 语言，C++ 则更为强调类型，其目的是为了在构建复杂的软件系统时，能够尽可能地在编译时期找到错误并提醒程序员。虽然 C++98 对于类型系统的构建已经近乎完美，却还是有枚举这样的漏网之鱼。所以 C++11 对其进行了增强。另外一方面，指针使用的安全则一直是 C++ 的重要议题。几乎所有的 C++ 的书籍都少不了指针方面的探讨。而 C++11 则再次为指针安全使用作出了努力。在本章，我们会看到 C++11 的做法。

5.1 强类型枚举

☞ **类别：部分人**

5.1.1 枚举：分门别类与数值的名字

枚举类型是 C 及 C++ 中一个基本的内置类型，不过也是一个有点“奇怪”的类型。从枚举的本意上来讲，就是要定义一个类别，并穷举同一类别下的个体以供代码中使用。由于枚举来源于 C，所以出于设计上的简单的目的，枚举值常常是对应到整型数值的一些名字。比如：

```
enum Gender { Male, Female };
```

定义了 Gender（性别）枚举类型，其中包含两种枚举值 Male 及 Famale。编译器会默认为 Male 赋值 0，为 Famale 赋值 1。这是 C 对名称的简单包装，即将名称对应到数值。

而枚举类型也可以是匿名的，匿名的枚举会有意想不到的用处。比如当程序中需要“数值的名字”的时候，我们常常可以使用以下 3 种方式来实现。

第一种方式是宏，比如：

```
#define Male 0
#define Female 1
```

宏的弱点在于其定义的只是预处理阶段的名字，如果代码中有 Male 或者 Female 的字符串，无论在什么位置一律将被替换。这在有的时候会干扰到正常的代码，因此很多时候为了避免这种情况，程序员会让宏全部以大写字母来命名，以区别于正常的代码。

而第二种方式——匿名的 enum 的状况会好些。

```
enum { Male, Female };
```

这里的匿名枚举中的 Male 和 Female 都是编译时期的名字，会得到编译器的检查。相比于宏的实现，匿名枚举不会有干扰正常代码的尴尬。

不过在 C++ 中，更受推荐是第三种方式——静态常量。如：

```
const static int Male = 0;
const static int Female = 1;
```

Male 和 Female 的名字同样得到编译时期检查。由于是静态常量，其名字作用域也被很好地局限于文件内。不过相比于 enum，静态常量不仅仅是一个编译时期的名字，编译器还可能会为 Male 和 Female 在目标代码中产生实际的数据，这会增加一点存储空间。相比而言，匿名的枚举似乎更为好用。

不过事实上，这 3 种“数值的名字”的实现方式孰优孰劣，程序员们各执一词。不过枚举类型的使用的独特性则是无需质疑的。

注意 历史上，枚举还有一个被称为“enum hack”的独特应用，在上面的静态常量的例子中，如果 static 的 Male 和 Female 声明在 class 中，在一些较早的编译器上不能为其就地赋值（赋值需要在 class 外），因此有人也采用了 enum 的方式在 class 中来代替常量声明。这就是“enum hack”。

虽然 enum 确实有些“奇怪”的用途，不过作为“枚举类型”本身而言，enum 并非完美，具体见下节。

5.1.2 有缺陷的枚举类型

C/C++ 的 enum 有个很“奇怪”的设定，就是具名（有名字）的 enum 类型的名字，以及 enum 的成员的名字都是全局可见的。这与 C++ 中具名的 namespace、class/struct 及 union 必须通过“名字 :: 成员名”的方式访问相比是格格不入的（namespace 等被称为强作用域类型，而 enum 则是非强作用域类型）。一不小心，程序员就容易遇到问题。比如下面两个枚举：

```
enum Type { General, Light, Medium, Heavy };
enum Category { General, Pistol, MachineGun, Cannon };
```

Category 中的 General 和 Type 中的 General 都是全局的名字，因此编译会报错。

而在下面的代码清单 5-1 中，则是一个通过 namespace 分割了全局空间，但 namespace 中的成员依然会被 enum 成员污染的例子。

代码清单 5-1

```
#include <iostream>
using namespace std;

namespace T{
    enum Type { General, Light, Medium, Heavy };
}

namespace {
    enum Category { General = 1, Pistol, MachineGun, Cannon };
}

int main() {
    T::Type t = T::Light;
    if (t == General)    // 忘记使用 namespace
        cout << "General Weapon" << endl;
    return 0;
}

// 编译选项 :g++ 5-1-1.cpp
```

可以看到，Category 在一个匿名 namespace 中，所以所有枚举成员名都默认进入全局名字空间。一旦程序员在检查 t 的值的时候忘记使用了 namespace T，就会导致错误的结果（事实上，有的编译器会在这里做出一些警告，但并不会阻止编译，而有的编译器则不会警告）。

另外，由于 C 中枚举被设计为常量数值的“别名”的本性，所以枚举的成员总是可以被隐式地转换为整型。很多时候，这也是不安全的。我们可以看看代码清单 5-2 所示的这个恼人的例子。

代码清单 5-2

```
#include <iostream>
using namespace std;

enum Type { General, Light, Medium, Heavy };
//enum Category { General, Pistol, MachineGun, Cannon }; // 无法编译通过，重复定义了 General
enum Category { Pistol, MachineGun, Cannon };

struct Killer {
    Killer(Type t, Category c) : type(t), category(c){}
    Type type;
    Category category;
};

int main() {
    Killer cool(General, MachineGun);
    // ...
```

```
    // ...其他很多代码...
    // ...
    if (cool.type >= Pistol)
        cout << "It is not a pistol" << endl;
    // ...
    cout << is_pod<Type>::value << endl;          // 1
    cout << is_pod<Category>::value << endl;      // 1

    return 0;
}

// 编译选项:g++ -std=c++11 5-1-2.cpp
```

在上面代码清单 5-2 的例子中，类型 Killer 同时拥有 Type 和 Category 两种命名类似的枚举类型成员。在一定时候，程序员想查看这位“冷酷”（cool）的“杀手”（Killer）是属于什么 Category 的。但很明显，程序员错用了成员 type。这是由于枚举类型数值在进行数值比较运算时，首先被隐式地提升为 int 类型数据，然后自由地进行比较运算。当然，程序员的本意并非如此（事实上，我们实验机上的编译器会给出警告说不同枚举类型枚举成员间进行了比较。但程序还是编译通过了，因为标准并不阻止这一点）。

为了解决这一问题，目前程序员一般会对枚举类型进行封装。可以看看代码清单 5-2 改良后的版本，如代码清单 5-3 所示。

代码清单 5-3

```
#include <iostream>
using namespace std;

class Type {
public:
    enum type { general, light, medium, heavy };
    type val;
public:
    Type(type t): val(t){}
    bool operator >= (const Type & t) { return val >= t.val; }
    static const Type General, Light, Medium, Heavy;
};

const Type Type::General(Type::general);
const Type Type::Light(Type::light);
const Type Type::Medium(Type::medium);
const Type Type::Heavy(Type::heavy);

class Category {
public:
    enum category { pistol, machineGun, cannon };
    category val;
```

```
public:
    Category(category c): val(c) {}
    bool operator >= (const Category & c) { return val >= c.val; }
    static const Category Pistol, MachineGun, Cannon;
};

const Category Category::Pistol(Category::pistol);
const Category Category::MachineGun(Category::machineGun);
const Category Category::Cannon(Category::cannon);

struct Killer {
    Killer(Type t, Category c) : type(t), category(c){}
    Type type;
    Category category;
};

int main() {
    // 使用类型包装后的 enum
    Killer notCool(Type::General, Category::MachineGun);
    // ...
    // ... 其他很多代码 ...
    // ...
    if (notCool.type >= Type::General)    // 可以通过编译
        cout << "It is not general" << endl;
    if (notCool.type >= Category::Pistol)    // 该句无法编译通过
        cout << "It is not a pistol" << endl;
// ...
    cout << is_pod<Type>::value << endl;          // 0
    cout << is_pod<Category>::value << endl;      // 0
    return 0;
}

// 编译选项 :g++ -std=c++11 5-1-3.cpp
```

封装的代码长得让人眼花缭乱，不过简单地说，封装即是使得枚举成员成为 class 的静态成员。由于 class 中的数据不会被默认转换为整型数据（除非定义相关操作符函数），所以可以避免被隐式转换。而且我们也可以看到，通过封装，枚举的成员也不再会污染全局名字空间了，使用时还必须带上 class 的名字，这样一来，之前枚举的一些小毛病都能够得到克服。

不过这种解决方案并非完美，至少可能有三个缺点：

- 显然，一般程序员不会为了简单的 enum 声明做这么复杂的封装。
- 由于封装且采用了静态成员，原本属于 POD 的 enum 被封装成为非 POD 的了（is_pod 均返回为 0，请对照代码清单 5-2 所示的情况），这会导致一系列的损失（参见 3.6 节）。
- 大多数系统的 ABI 规定，传递参数的时候如果参数是个结构体，就不能使用寄存器来传参（只能放在堆栈上），而相对地，整型可以通过寄存器中传递。所以一旦将 class

封装版本的枚举作为函数参数传递，就可能带来一定的性能损失。

无论上述哪一条，对于封装方案来说都是极为不利的。

此外，枚举类型所占用的空间大小也是一个“不确定量”。标准规定，C++ 枚举所基于的“基础类型”是由编译器来具体指定实现的，这会导致枚举类型成员的基本类型的不确定性问题（尤其是符号性）。我们可以看看代码清单 5-4 所示的这个例子。

代码清单 5-4

```
#include <iostream>
using namespace std;

enum C { C1 = 1, C2 = 2};
enum D { D1 = 1, D2 = 2, Dbig = 0xFFFFFFF0U };
enum E { E1 = 1, E2 = 2, Ebig = 0xFFFFFFFFLL};
int main() {
    cout << sizeof(C1) << endl;    // 4

    cout << Dbig << endl;    // 编译器输出不同，g++: 4294967280
    cout << sizeof(D1) << endl;      // 4
    cout << sizeof(Dbig) << endl;    // 4

    cout << Ebig << endl;    // 68719476735
    cout << sizeof(E1) << endl; // 8
    return 0;
}

// 编译选项 :g++ 5-1-4.cpp
```

在代码清单 5-4 所示的例子当中，我们可以看到，编译器会根据数据类型的不同对 enum 应用不同的数据长度。在我们对 g++ 的测试中，普通的枚举使用了 4 字节的内存，而当需要的时候，会拓展为 8 字节。此外，对于不同的编译器，上例中 Dbig 的输出结果将会不同：使用 Visual C++ 编译程序的输出结果为 –16，而使用 g++ 来编译输出为 4294967280。这是由于 Visual C++ 总是使用无符号类型作为枚举的底层实现，而 g++ 会根据枚举的类型进行变动造成的。

5.1.3 强类型枚举以及 C++11 对原有枚举类型的扩展

非强类型作用域，允许隐式转换为整型，占用存储空间及符号性不确定，都是枚举类的缺点。针对这些缺点，新标准 C++11 引入了一种新的枚举类型，即“枚举类”，又称“强类型枚举”(strong-typed enum)。

声明强类型枚举非常简单，只需要在 enum 后加上关键字 class。比如：

```
enum class Type { General, Light, Medium, Heavy };
```

就声明了一个强类型的枚举 Type。强类型枚举具有以下几点优势：

- 强作用域，强类型枚举成员的名称不会被输出到其父作用域空间。
- 转换限制，强类型枚举成员的值不可以与整型隐式地相互转换。
- 可以指定底层类型。强类型枚举默认的底层类型为 int, 但也可以显式地指定底层类型，具体方法为在枚举名称后面加上“：type”，其中 type 可以是除 wchar_t 以外的任何整型。比如：

```
enum class Type: char { General, Light, Medium, Heavy };
```

就指定了 Type 是基于 char 类型的强类型枚举。

我们可以看看具体的例子，如代码清单 5-5 所示。

代码清单 5-5

```
#include <iostream>
using namespace std;

enum class Type { General, Light, Medium, Heavy };
enum class Category { General = 1, Pistol, MachineGun, Cannon };

int main() {
    Type t = Type::Light;
    t = General;                        // 编译失败，必须使用强类型名称
    if (t == Category::General)         // 编译失败，必须使用 Type 中的 General
        cout << "General Weapon" << endl;
    if (t > Type::General)              // 通过编译
        cout << "Not General Weapon" << endl;
    if (t > 0)                          // 编译失败，无法转换为 int 类型
        cout << "Not General Weapon" << endl;
    if ((int)t > 0)                     // 通过编译
        cout << "Not General Weapon" << endl;
    cout << is_pod<Type>::value << endl;          // 1
    cout << is_pod<Category>::value << endl;      // 1
    return 0;
}

// 编译选项 :g++ -std=c++11 5-1-5.cpp
```

在代码清单 5-5 中，我们定义了两个强类型枚举 Type 和 Category，它们都包含一个称为 General 的成员。由于强类型枚举成员的名字不会输出到父作用域，因此不会有编译问题。也由于不输出成员名字，所以我们在使用该类型成员的时候必须加上其所属的枚举类型的名字。此外，可以看到，枚举成员间仍然可以进行数值式的比较，但不能够隐式地转为 int 型。事实上，如果要将强类型枚举转化为其他类型，必须进行显式转换。

事实上，强类型制止 enum 成员和 int 之间的转换，使得枚举更加符合“枚举”的本来意

义，即对同类进行列举的一个集合，而定义其与数值间的关联则使之能够默认拥有一种对成员排列的机制。而制止成员名字输出则进一步避免了名字空间冲突的问题。这两点跟之前我们使用 class 对枚举进行封装并无二致。不过新的强类型枚举没有任何 class 封装枚举的缺点。我们可以看到，Type 和 Category 都是 POD 类型，不会像 class 封装版本一样被编译器视为结构体，书写也很简便。在拥有类型安全和强作用域两重优点的情况下，几乎没有任何额外的开销。

此外，由于可以指定底层基于的基本类型，我们可以避免编译器不同而带来的不可移植性。此外，设置较小的基本类型也可以节省内存空间，如代码清单 5-6 所示。

代码清单 5-6

```
#include <iostream>
using namespace std;

enum class C : char { C1 = 1, C2 = 2};
enum class D : unsigned int { D1 = 1, D2 = 2, Dbig = 0xFFFFFFF0U };

int main() {
    cout << sizeof(C::C1) << endl;    // 1

    cout << (unsigned int)D::Dbig << endl;    // 编译器输出一致,4294967280
    cout << sizeof(D::D1) << endl;      // 4
    cout << sizeof(D::Dbig) << endl;    // 4
    return 0;
}
// 编译选项 :g++ -std=c++11 5-1-6.cpp
```

在代码清单 5-6 中，我们为强类型枚举 C 指定底层基本类型为 char，因为我们只有 C1、C2 两个值较小的成员，一个 char 足以保存所有的枚举成员。而对于强类型枚举 D，我们指定基本类型为 unsigned int，则所有编译器都会使用无符号的 unsigned int 来保存该枚举。故各个编译器都能保证一致的输出。

相比于原来的枚举，强类型枚举更像是一个属于 C++ 的枚举。但为了配合新的枚举类型，C++11 还对原有枚举类型进行了扩展。

首先是底层的基本类型方面。在新标准 C++11 中，原有枚举类型的底层类型在默认情况下，仍然由编译器来具体指定实现。但也可以跟强类型枚举类一样，显式地由程序员来指定。其指定的方式跟强类型枚举一样，都是枚举名称后面加上“：type”，其中 type 可以是除 wchar_t 以外的任何整型。比如：

```
enum Type: char { General, Light, Medium, Heavy };
```

在 C++11 中也是一个合法的 enum 声明。

第二个扩展则是作用域的。在 C++11 中，枚举成员的名字除了会自动输出到父作用域，

也可以在枚举类型定义的作用域内有效。比如：

```
enum Type { General, Light, Medium, Heavy };
Type t1 = General;
Type t2 = Type::General;
```

General 和 Type::General 两行都是合法的使用形式。

这两个扩展都保留了向后兼容性，也方便了程序员在代码中同时操作两种枚举类型。

此外，我们在声明强类型枚举的时候，也可以使用关键字 enum struct。事实上 enum struct 和 enum class 在语法上没有任何区别（enum class 的成员没有公有私有之分，也不会使用模板来支持泛化的声明）。

有一点比较有趣的是匿名的 enum class。由于 enum class 是强类型作用域的，故匿名的 enum class 很可能什么都做不了，如代码清单 5-7 所示。

代码清单 5-7

```
enum class { General, Light, Medium, Heavy } weapon;

int main() {
    weapon = General;    // 无法编译通过
    bool b = (weapon == weapon::General);    // 无法编译通过
    return 0;
}

// 编译选项 :g++ -std=c++11 5-1-7.cpp
```

代码清单 5-7 中我们声明了一个匿名的 enum class 实例 weapon，却无法对其设置值或者比较其值（这和匿名 struct 是不一样的）。事实上，使用 enum class 的时候，应该总是为 enum class 提供一个名字（我们实验机上的 clang 编译器以及 XLC 编译器甚至会因为用户使用匿名的强类型枚举而阻止编译）。联系到我们在 5.1.1 中提到的让匿名 enum 成为“数值的名字”，匿名的 enum class 则完全做不到。所以在实际使用中必须注意（当然，程序员还是可以通过 decltype 来获得匿名强类型枚举的类型并且进行使用，即使这样做没什么太大的意义，请参见 4.3 节）。

5.2 堆内存管理：智能指针与垃圾回收

☞ 类别：类作者、库作者

5.2.1 显式内存管理

程序员在处理现实生活中的 C/C++ 程序的时候，常会遇到诸如程序运行时突然退出，或

占用的内存越来越多，最后不得不定期重启的一些典型症状。这些问题的源头可以追溯到 C/C++ 中的显式堆内存管理上。通常情况下，这些症状都是由于程序没有正确处理堆内存的分配与释放造成的，从语言层面来讲，我们可以将其归纳为以下一些问题。

- 野指针：一些内存单元已被释放，之前指向它的指针却还在被使用。这些内存有可能被运行时系统重新分配给程序使用，从而导致了无法预测的错误。
- 重复释放：程序试图去释放已经被释放过的内存单元，或者释放已经被重新分配过的内存单元，就会导致重复释放错误。通常重复释放内存会导致 C/C++ 运行时系统打印出大量错误及诊断信息。
- 内存泄漏：不再需要使用的内存单元如果没有被释放就会导致内存泄漏。如果程序不断地重复进行这类操作，将会导致内存占用剧增。

虽然显式的管理内存在性能上有一定的优势，但也被广泛地认为是容易出错的。随着多线程程序的出现和广泛使用，内存管理不佳的情况还可能会变得更加严重。因此，很多程序员也认为编程语言应该提供更好的机制，让程序员摆脱内存管理的细节。在 C++ 中，一个这样的机制就是标准库中的智能指针。在 C++11 新标准中，智能指针被进行了改进，以更加适应实际的应用需求。而进一步地，标准库还提供了所谓“最小垃圾回收”的支持。

5.2.2 C++11 的智能指针

在 C++98 中，智能指针通过一个模板类型“auto_ptr”来实现。auto_ptr 以对象的方式管理堆分配的内存，并在适当的时间（比如析构），释放所获得的堆内存。这种堆内存管理的方式只需要程序员将 new 操作返回的指针作为 auto_ptr 的初始值即可，程序员不用再显式地调用 delete。比如：

```
auto_ptr(new int);
```

这在一定程度上避免了堆内存忘记释放而造成的问题。不过 auto_ptr 有一些缺点（拷贝时返回一个左值，不能调用 delete [] 等），所以在 C++11 标准中被废弃了。C++11 标准中改用 unique_ptr、shared_ptr 及 weak_ptr 等智能指针来自动回收堆分配的对象。

这里我们可以看一个 C++11 中使用新的智能指针的简单例子，如代码清单 5-8 所示。

代码清单 5-8

```
#include <memory>
#include <iostream>
using namespace std;

int main() {
    unique_ptr<int> up1(new int(11));    // 无法复制的 unique_ptr
    unique_ptr<int> up2 = up1;           // 不能通过编译
```

```
    cout << *up1 << endl;    // 11

    unique_ptr<int> up3 = move(up1);     // 现在 p3 是数据唯一的 unique_ptr 智能指针

    cout << *up3 << endl;    // 11
    cout << *up1 << endl;    // 运行时错误
    up3.reset();             // 显式释放内存
    up1.reset();             // 不会导致运行时错误
    cout << *up3 << endl;    // 运行时错误

    shared_ptr<int> sp1(new int(22));
    shared_ptr<int> sp2 = sp1;

    cout << *sp1 << endl;    // 22
    cout << *sp2 << endl;    // 22

    sp1.reset();
    cout << *sp2 << endl;    // 22
}

// 编译选项 :g++ -std=c++11 5-2-1.cpp
```

在代码清单 5-8 中，使用了两种不同的智能指针 unique_ptr 及 shared_ptr 来自动地释放堆对象的内存。由于每个智能指针都重载了 * 运算符，用户可以使用 *up1 这样的方式来访问所分配的堆内存。而在该指针析构或者调用 reset 成员的时候，智能指针都可能释放其拥有的堆内存。从作用上来讲，unique_ptr 和 shared_ptr 还是和以前的 auto_ptr 保持了一致。

不过从代码清单 5-8 中还是可以看到，unique_ptr 和 shared_ptr 在对所占内存的共享上还是有一定区别的。

直观地看来，unique_ptr 形如其名地，与所指对象的内存绑定紧密，不能与其他 unique_ptr 类型的指针对象共享所指对象的内存。比如，本例中的 unique_ptr<int> up2 = up1; 不能通过编译，是因为每个 unique_ptr 都是唯一地“拥有”所指向的对象内存，由于 up1 唯一地占有了 new 分配的堆内存，所以 up2 无法共享其“所有权”。事实上，这种“所有权”仅能够通过标准库的 move 函数来转移。我们可以看到代码中 up3 的定义，unique_ptr<int> up3 = move(up1); 一旦“所有权”转移成功了，原来的 unique_ptr 指针就失去了对象内存的所有权。此时再使用已经“失势”的 unique_ptr，就会导致运行时的错误。本例中的后段使用 *up1 就是很好的例子。

而从实现上讲，unique_ptr 则是一个删除了拷贝构造函数、保留了移动构造函数的指针封装类型（我们在 7.2 节中可以看到如何删除一个类的拷贝构造函数）。程序员仅可以使用右值对 unique_ptr 对象进行构造，而且一旦构造成功，右值对象中的指针即被“窃取”，因此该右值对象即刻失去了对指针的“所有权”。

而 shared_ptr 同样形如其名，允许多个该智能指针共享地“拥有”同一堆分配对象的内存。与 unique_ptr 不同的是，由于在实现上采用了引用计数，所以一旦一个 shared_ptr 指针放弃了“所有权”（失效），其他的 shared_ptr 对对象内存的引用并不会受到影响。代码清单 5-8 中，智能指针 sp2 就很好地说明了这种状况。虽然 sp1 调用了 reset 成员函数，但由于 sp1 和 sp2 共享了 new 分配的堆内存，所以 sp1 调用 reset 成员函数只会导致引用计数的降低，而不会导致堆内存的释放。只有在引用计数归零的时候，share_ptr 才会真正释放所占有的堆内存的空间。

在 C++11 标准中，除了 unique_ptr 和 shared_ptr，智能指针还包括了 weak_ptr 这个类模板。weak_ptr 的使用更为复杂一点，它可以指向 shared_ptr 指针指向的对象内存，却并不拥有该内存。而使用 weak_ptr 成员 lock，则可返回其指向内存的一个 shared_ptr 对象，且在所指对象内存已经无效时，返回指针空值（nullptr，请参见 7.1 节）。这在验证 share_ptr 智能指针的有效性上会很有作用，如代码清单 5-9 所示。

代码清单 5-9

```
#include <memory>
#include <iostream>
using namespace std;

void Check(weak_ptr<int> & wp) {
    shared_ptr<int> sp = wp.lock(); // 转换为 shared_ptr<int>
    if (sp != nullptr)
        cout << "still " << *sp << endl;
    else
        cout << "pointer is invalid." << endl;
}

int main() {
    shared_ptr<int> sp1(new int(22));
    shared_ptr<int> sp2 = sp1;
    weak_ptr<int> wp = sp1; // 指向 shared_ptr<int> 所指对象

    cout << *sp1 << endl;   // 22
    cout << *sp2 << endl;   // 22
    Check(wp);              // still 22

    sp1.reset();
    cout << *sp2 << endl;   // 22
    Check(wp);              // still 22

    sp2.reset();
    Check(wp);              // pointer is invalid
}

// 编译选项 :g++ -std=c++11 5-2-2.cpp
```

在代码清单 5-9 中，我们定义了一个共享对象内存的两个 shared_ptr——sp1 及 sp2。而 weak_ptr wp 同样指向该对象内存。可以看到，在 sp1 及 sp2 都有效的时候，我们调用 wp 的 lock 函数，将返回一个有效的 shared_ptr 对象供使用，于是 Check 函数会输出以下内容：

```
still 22
```

此后我们分别调用了 sp1 及 sp2 的 reset 函数，这会导致对唯一的堆内存对象的引用计数降至 0。而一旦引用计数归 0，shared_ptr<int> 就会释放堆内存空间，使之失效。此时我们再调用 weak_ptr 的 lock 函数时，则返回一个指针空值 nullptr。这时 Check 函数则会打印出：

```
pointer is invalid
```

这样的语句了。在整个过程当中，只有 shared_ptr 参与了引用计数，而 weak_ptr 没有影响其指向的内存的引用计数。因此可以验证 shared_ptr 指针的有效性。

简单情况下，程序员用 unique_ptr 代替以前使用 auto_ptr 的代码就可以使用 C++11 中的智能指针。而 shared_ptr 及 weak_ptr 则可用在用户需要引用计数的地方。事实上，关于智能指针的历史、使用及各种讨论还有很多，不过本书的重点不在于标准库，因此这里就不一一展开了。

总地来说，虽然智能指针能帮助用户进行有效的堆内存管理，但是它还是需要程序员显式地声明智能指针，而完全不需要考虑回收指针类型的内存管理方案可能会更讨人喜欢。当然，这种方案早已有了，就是垃圾回收机制。

5.2.3 垃圾回收的分类

如果追根溯源的话，显式内存管理的替代方案很早就有了。因为早在 1959 年前后，约翰·麦肯锡（John McCarthy）就为 Lisp 语言发明了所谓“垃圾回收”的方法。这里，我们把之前使用过，现在不再使用或者没有任何指针再指向的内存空间就称为“垃圾”。而将这些“垃圾”收集起来以便再次利用的机制，就被称为“垃圾回收”（Garbage Collection）。在编程语言的发展过程中，垃圾回收的堆内存管理也得到了很大的发展。如今，垃圾回收机制已经大行其道，在大多数编程语言中，我们都可以看到对垃圾回收特性的支持，如表 5-1 所示。

表 5-1 各种编程语言对垃圾回收的支持情况

编程语言	对垃圾回收的支持情况
C++	部分
Java	支持
Python	支持
C	不支持
C#	支持

（续）

编程语言	对垃圾回收的支持情况
Ruby	支持
PHP	支持
Perl	支持
Hashkell	支持
Pascal	不支持

垃圾回收的方式虽然很多，但主要可以分为两大类：

1. 基于引用计数（reference counting garbage collector）的垃圾回收器

简单地说，引用计数主要是使用系统记录对象被引用（引用、指针）的次数。当对象被引用的次数变为0时，该对象即可被视作“垃圾”而回收。使用引用计数做垃圾回收的算法的一个优点是实现很简单，与其他垃圾回收算法相比，该方法不会造成程序暂停，因为计数的增减与对象的使用是紧密结合的。此外，引用计数也不会对系统的缓存或者交换空间造成冲击，因此被认为“副作用”较小。但是这种方法比较难处理“环形引用”问题，此外由于计数带来的额外开销也并不小，所以在实用上也有一定的限制。

2. 基于跟踪处理（tracing garbage collector）的垃圾回收器

相比于引用计数，跟踪处理的垃圾回收机制被更为广泛地应用。其基本方法是产生跟踪对象的关系图，然后进行垃圾回收。使用跟踪方式的垃圾回收算法主要有以下几种：

（1）标记－清除（Mark-Sweep）

顾名思义，这个算法可以分为两个过程。首先该算法将程序中正在使用的对象视为“根对象”，从根对象开始查找它们所引用的堆空间，并在这些堆空间上做标记。当标记结束后，所有被标记的对象就是可达对象（Reachable Object）或活对象（Live Object），而没有被标记的对象就被认为是垃圾，在第二步的清扫（Sweep）阶段会被回收掉。

这种方法的特点是活的对象不会被移动，但是其存在会出现大量的内存碎片的问题。

（2）标记－整理（Mark-Compact）

这个算法标记的方法和标记-清除方法一样，但是标记完之后，不再遍历所有对象清扫垃圾了，而是将活的对象向“左”靠齐，这就解决了内存碎片的问题。

标记－整理的方法有个特点就是移动活的对象，因此相应的，程序中所有对堆内存的引用都必须更新。

（3）标记－拷贝（Mark-Copy）

这种算法将堆空间分为两个部分：From 和 To。刚开始系统只从 From 的堆空间里面分配

内存，当 From 分配满的时候系统就开始垃圾回收：从 From 堆空间找出所有活的对象，拷贝到 To 的堆空间里。这样一来，From 的堆空间里面就全剩下垃圾了。而对象被拷贝到 To 里之后，在 To 里是紧凑排列的。接下来是需要将 From 和 To 交换一下角色，接着从新的 From 里面开始分配。

标记 - 拷贝算法的一个问题是堆的利用率只有一半，而且也需要移动活的对象。此外，从某种意义上讲，这种算法其实是标记 - 整理算法的另一种实现而已。

虽然历来 C++ 都没有公开地支持过垃圾回收机制，但垃圾回收并非某些语言的专利。事实上，C++11 标准也开始对垃圾回收做了一定的支持，虽然支持的程度还非常有限，但我们已经看到了 C++ 语言变得更为强大的端倪。

5.2.4 C++ 与垃圾回收

如我们提到的，在 C++11 中，智能指针等可以支持引用计数。不过由于引用计数并不能有效解决形如“环形引用”等问题，其使用会受到一些限制。而且基于一些其他的原因，比如因多线程程序等而引入的内存管理上的困难，程序员可能也会需要垃圾回收。

一些第三方的 C/C++ 库已经支持标记 - 清除方法的垃圾回收，比如一个比较著名的 C/C++ 垃圾回收库——Boehm [⊖]。该垃圾回收器需要程序员使用库中的堆内存分配函数显式地替代 malloc，继而将堆内存的管理交给垃圾回收器来完成垃圾回收。不过由于 C/C++ 中指针类型的使用非常灵活，这样的库在实际使用中会有一些限制，可移植性也不好。

为了解决垃圾回收中的安全性和可移植性问题，在 2007 年，惠普的 Hans-J. Boehm（Boehm 的作者）和赛门铁克的 Mike Spertus 共同向 C++ 委员会递交了一个关于 C++ 中垃圾回收的提案。该提案通过添加 gc_forbidden、gc_relaxed、gc_required、gc_safe、gc_strict 等关键字来支持 C++ 语言中的垃圾回收。该提案甚至可以让程序员显式地要求垃圾回收。刚开始这得到了大多数委员的支持，后来却在标准的初稿中删除了，原因是该特性过于复杂，并且还存在一些问题（比如与显式调用析构函数的现有的库的兼容问题等）。所以，Boehm 和 Spertus 对初稿进行了简化，仅仅保留了支持垃圾回收的最基本的部分，即通过对语言的约束，来保证安全的垃圾回收。这也是我们现在看到的 C++11 标准中的“最小垃圾回收支持”的历史来由。

而要保证安全的垃圾回收，首先必须知道 C/C++ 语言中什么样的行为可能导致垃圾回收中出现“不安全”的状况。简单地说，不安全源自于 C/C++ 语言对指针的“放纵”，即允许过分灵活的使用。我们可以看代码清单 5-10 所示的例子。

⊖ http://www.hpl.hp.com/personal/Hans_Boehm/gc/

代码清单 5-10

```
int main(){
    int* p = new int;
    p += 10;     // 移动指针，可能导致垃圾回收器
    p -= 10;     // 回收原来指向的内存
    *p = 10;     // 再次使用原本相同的指针则可能无效
}

// 编译选项 :g++ 5-2-3.cpp
```

在代码清单 5-10 中，我们对指针 p 做了自加和自减操作。这在 C/C++ 中被认为是合理的，因为指针有所指向的类型，自加或者自减能够使程序员轻松地找到“下一个”同样的对象（实际是一个迭代器的概念）。不过对于垃圾回收来说，一旦 p 指向了别的地址，则可认为 p 曾指向的内存不再使用。垃圾回收器可以据此对其进行回收。这对之后 p 的使用（*p = 10）带来的后果将是灾难性的。

我们再来看一个例子，如代码清单 5-11 所示。

代码清单 5-11

```
int main() {
    int *p = new int;
    int *q = (int*)(reinterpret_cast<long long>(p) ^ 2012);    // q 隐藏了 p
    // 做一些其他工作，垃圾回收器可能已经回收了 p 指向对象
    q = (int*)(reinterpret_cast<long long>(q) ^ 2012); // 这里的 q == p
    *q = 10;
}

// 编译选项 :g++ 5-2-4.cpp
```

在代码清单 5-11 中，我们用指针 q 隐藏了指针 p。而之后又用可逆的异或运算将 p “恢复”了出来。在 main 函数中，p 实际所指向的内存都是有效的，但由于该指针被隐藏了，垃圾回收器可以早早地将 p 指向的对象回收掉。同样，语句 *q = 10 的后果也是灾难性的。

指针的灵活使用可能是 C/C++ 中的一大优势，而对于垃圾回收来说，却会带来很大的困扰。被隐藏的指针会导致编译器在分析指针的可达性（生命周期）时出错。而即使编译器开发出了隐藏指针分析的手段，其带来的编译开销也不会让程序员对编译时间的显著增长视而不见。历史上，解决这种问题的方法通常是新接口。C++11 和垃圾回收的解决方案也不例外，就是让程序员利用这样的接口来提示编译器代码中存在指针不安全的区域。

5.2.5 C++11 与最小垃圾回收支持

C++11 新标准为了做到最小的垃圾回收支持，首先对“安全”的指针进行了定义，或者

使用 C++11 中的术语说，安全派生（safely derived）的指针。安全派生的指针是指向由 new 分配的对象或其子对象的指针。安全派生指针的操作包括：

- ❑在解引用基础上的引用，比如：&*p。
- ❑定义明确的指针操作，比如：p + 1。
- ❑定义明确的指针转换，比如：static_cast <void *> (p)。
- ❑指针和整型之间的 reinterpret_case，比如：reinterpret_cast <intptr_t>(p)。

注意 intptr_t 是 C++11 中一个可选择实现的类型，其长度等于平台上指针的长度（通过 decltype 声明）。

我们可以回头看看代码清单 5-11。reinterpret_cast<long long>(p) 是合法的安全派生操作，而转换后的指针再进行异或操作：reinterpret_cast<long long>(p) ^ 2012 之后，指针就不再是安全派生的了，这是因为异或操作（^）不是一个安全派生操作。同理，reinterpret_cast<long long> (q) ^ 2012 也不是安全派生指针。因此，根据定义，在使用内存回收器的情况下，*q = 10 的行为是不确定的，如果程序在此处发生错误也是合理的。

在 C++11 的规则中，最小垃圾回收支持是基于安全派生指针这个概念的。程序员可以通过 get_pointer_safety 函数查询来确认编译器是否支持这个特性。get_pointer_safety 的原型如下：

```
pointer_safety get_pointer_safety() noexcept
```

其返回一个 pointer_safety 类型的值。如果该值为 pointer_safety::strict，则表明编译器支持最小垃圾回收及安全派生指针等相关概念，如果该值为 pointer_safety::relax 或是 pointer_safety::preferred，则表明编译器并不支持，基本上跟没有垃圾回收的 C 和 C++98 一样。不过按照一些解释，pointer_safety::preferred 和 pointer_safety::relax 也略有不同，前者垃圾回收器可能被用作一些辅助功能，如内存泄露检测或检测对象是否被一个错误的指针解引用（事实上，在本书编写时，几乎没有编译器实现了最小垃圾回收支持，甚至连 get_pointer_safety 这个函数接口都还没实现）。

此外，如果程序员代码中出现了指针不安全使用的状况，C++11 允许程序员通过一些 API 来通知垃圾回收器不得回收该内存。C++11 的最小垃圾回收支持使用了垃圾回收的术语，即需声明该内存为“可到达”的。

```
void declare_reachable( void* p );
template <class T> T *undeclare_reachable(T *p) noexcept;
```

declare_reachable() 显式地通知垃圾回收器某一个对象应被认为可达的，即使它的所有指针都对回收器不可见。undeclare_reachable() 则可以取消这种可达声明。针对代码清单 5-11，我们对隐藏的指针做一些声明，如代码清单 5-12 所示。

代码清单 5-12

```
#include <memory>
using namespace std;

int main() {
    int *p = new int;
    declare_reachable(p);    // 在 p 被隐藏之前声明为可达的
    int *q = (int*)((long long)p ^ 2012);
    // 解除可达声明
    q = undeclare_reachable<int>((int*)((long long)q ^ 2012));
    *q = 10;
}
```

代码清单 5-12 可能是一个能够运行的例子。这里，我们在 p 指针被不安全派生（隐藏）之前使用 declare_reachable 声明其是可达的。这样一来，它会被垃圾回收器忽略而不会被回收。而在我们通过可逆的异或运算使得 q 指针指向 p 所指对象时，我们则使用了 undeclare_reachable 来取消可达声明。注意 undeclare_reachable 不是通知垃圾回收器 p 所指对象已经可以回收。实际上，declare_reachable 和 undeclare_reachable 只是确立了一个代码范围，即在两者之间的代码运行中，p 所指对象不会被垃圾回收器所回收。

这里可能有的读者会注意到一个细节，declare_reachable 只需要传入一个简单的 void* 指针，但 undeclare_reachable 却被设计为一个函数模板。这是一个极不对称的设计。但事实上 undeclare_reachable 使用模板的主要目的是为了返回合适类型以供程序使用。而垃圾回收器本来就知道指针所指向的内存的大小，因此 declare_reachable 传入 void* 指针就已经足够了。

有的时候程序员会选择在一大片连续的堆内存上进行指针式操作，为了让垃圾回收器不关心该区域，也可以使用 declare_no_pointers 及 undeclare_no_pointers 函数来告诉垃圾回收器该内存区域不存在有效的指针。

```
void declare_no_pointers(char *p, size_t n) noexcept;
void undeclare_no_pointers(char *p, size_t n) noexcept;
```

其使用方式与 declare_reachable 及 undeclare_reachable 类似，不过指定的是从 p 开始的连续 n 的内存。

5.2.6 垃圾回收的兼容性

尽管在设计 C++11 标准时想尽可能保证向后兼容，但是对于垃圾回收来说，破坏向后兼容是不可避免的。通常情况下，如果我们想要程序使用垃圾回收，或者可靠的内存泄漏检测，我们就必须做出必要的假设来保证垃圾回收器能工作。而为此，我们必须限制指针的使用或者使用 declare_reachable/undeclare_reachable、declare_no_pointer/undeclare_no_pointer 来让一些不安全的指针使用免于垃圾回收器的检查。因此想让老的代码毫不费力地使用垃圾回

收，现实情况下对大多数代码还是不可能的。

此外，C++11 标准中对指针的垃圾回收支持仅限于系统提供的 new 操作符分配的内存，而 malloc 分配的内存则会被认为总是可达的，即无论何时垃圾回收器都不予回收。因此使用 malloc 等的较老代码的堆内存还是必须由程序员自己控制。

而更为现实的状况是在本书写作时，垃圾回收的特性还没有得到任何编译器支持，即使是所谓的“最小垃圾回收”。标准的发展以及垃圾回收在 C/C++ 中的实现可能还需要一定的时间。不过有了最小支持，用户可能在新代码中会注意指针的使用，并对形如指针隐藏的状况使用合适的函数来对被隐藏指针的堆对象进行保护。按照 C++ 的设计，显式地 delete 使用与垃圾回收并不会形成冲突。如果程序员选择这么做的话，就应该能够保证最大的代码向前兼容性。在未来某个时刻 C++ 垃圾回收支持完成的时候，代码可以直接享受其带来的益处。

5.3 本章小结

C++ 是一种静态类型的语言，在 C++ 的发展中，一系列的改进使得 C++ 的类型机制几乎严丝合缝，滴水不漏。而在 C++11 中，最后的漏网之鱼——枚举，终于也以新增的强类型枚举的方式进行了规范。虽然为了兼容性，旧的枚举语义没有任何改变，但新的强类型枚举可以避免形如名字冲突、隐式向整型转换等诸多问题，所以在使用上更加安全。事实上，现有的标准库中的枚举就已经大量采用了强类型枚举来规避编程中的风险。

而堆分配变量的自动释放从来都是编程者津津乐道的问题。理想情况下，程序员应该总是在栈上分配变量，这样变量能够有效地自动释放。不过现实情况下，很多时候程序员在编程时并不能预期所需内存的使用期，所以程序员还离不开 malloc/free 或者 new/delete 的堆式内存分配。在 C++98 中，用户可以使用智能指针 auto_ptr 来自动释放内存。C++11 中则进一步改进了 auto_ptr——程序员可以使用行为更加良好的 unique_ptr 和 shared_ptr/weak_ptr 智能指针来进行自动的堆内存释放。虽然 unique/shared/weak 在概念上比起简单的 auto 变得复杂了，但是行为上却更加安全（返回右值、调用 delete[] 等）。程序员应该在 C++11 中全面使用新的智能指针代替 auto_ptr。

而随着编程模型的复杂化，C++ 语言也在考虑引入全面的垃圾回收。不过在 C++11 中，全面的垃圾回收这个心愿尚未达成。实际在 C++11 中的是“最小化”的垃圾回收支持，即定义了什么样的指针对于垃圾回收是安全的，什么样的动作会对垃圾回收造成影响。一旦有了会对垃圾回收造成影响的操作，程序员则可以调用 API 来通知垃圾回收器代码对于垃圾回收是否可达。事实上，现有情况下，我们并看不到“最小垃圾回收支持”的实质应用，大多数编译器也还没有实现相关的内容，但有了最小的支持，程序员和编译器及库的制作者间的接口也就清晰了。不排除以后有编译器或者库制作者设计出能够实现垃圾回收的、能够编译满足最小垃圾回收支持的 C++ 代码的编译器或库产品。不过，能否实现还需要假以时日。

第⑥章

提高性能及操作硬件的能力

对于用 C++ 编写的程序来说，性能是永恒的主题。相比于一些流行的高级语言，C++ 程序常会具有不可比拟的性能优势。在 C++11 中，程序员还能够进一步挖掘程序运行性能。而面对硬件系统的日新月异，尤其是多核多线程编程的浪潮，C++11 依然能横刀立马，通过对底层的硬件操作进行良好的抽象，充分满足了程序员利用多核多线程性能的需求。通过阅读本章，读者可以了解 C++11 在性能、硬件操作上的各种创新。极有可能，C++11 将引领新的并行编程的浪潮。

6.1 常量表达式

☞ 类别：类作者

6.1.1 运行时常量性与编译时常量性

在 C++ 中，我们常常会遇到常量的概念。常量表示该值不可修改，通常是通过 const 关键字来修饰的。比如：

```
const int i = 3;
```

上述代码就声明了一个名字为 i 的常量。const 还可以修饰函数参数、函数返回值、函数本身、类等。在不同的使用条件下，const 有不同的意义，不过大多数情况下，const 描述的都是一些“运行时常量性”的概念，即具有运行时数据的不可更改性。不过有的时候，我们需要的却是编译时期的常量性，这是 const 关键字无法保证的。我们可以看看代码清单 6-1 所示的例子。

代码清单 6-1

```
const int GetConst() { return 1; }

void Constless(int cond) {
    int arr[GetConst()] = {0};      // 无法通过编译

    enum { e1 = GetConst(), e2 };   // 无法通过编译

    switch (cond) {
```

```
        case GetConst():                    // 无法通过编译
            break;
        default:
            break;
    }
}

// 编译选项 :g++ -c 6-1-1.cpp
```

在代码清单 6-1 中，我们定义了一个返回常数 1 的函数 GetConst。我们使用了 const 关键字修饰了返回类型。不过编译后我们发现，无论将 GetConst 的结果用于需要初始化数组 Arr 的声明中，还是用于匿名枚举中，或用于 switch-case 的 case 表达式中，编译器都会报告错误。发生这样错误的原因如我们上面提到的一样，这些语句都需要的是编译时期的常量值。而 const 修饰的函数返回值，只保证了在运行时期内其值是不可以被更改的。这是两个完全不同的概念。

我们再来看一个 BitSet 的例子，该例子更贴近开发人员的实际状况，如代码清单 6-2 所示。

代码清单 6-2

```
enum BitSet {
    V0   =   1 << 0,
    V1   =   1 << 1,
    V2   =   1 << 2,
    VMAX =   1 << 3
};

// 重定义操作符 "|", 以保证返回的 BitSet 值不超过枚举的最大值
const BitSet operator|(BitSet x, BitSet y) {
    return static_cast<BitSet>(((int)x | y) & (VMAX - 1));
}

template <int i = V0 | V1>  // 无法通过编译
    void LikeConst(){}

// 编译选项 :clang++ -c -std=c++11 6-1-2.cpp
```

在代码清单 6-2 中，我们看到一个有限成员的 BitSet 的枚举类型的定义（读者是否还记得代码清单 2-7 所示的例子）。而为了尽可能地保证或操作的有效性，我们重载了 operator，该操作除了进行或运算外，还会通过 &(VMAX - 1) 这样的操作保证该或操作的输出不会超过 VMAX 枚举值。而此时我们将 V0 | V1 作为非类型模板函数的默认模板参数，则会导致编译错误。这同样是由需要的是编译时常量所导致的。

那么有没有什么办法可以通过更改上面的例子，获得编译时期的常量呢？最简单的方法

是，我们可以在代码清单 6-1 的文件开始使用 C 中的宏替代 GetConst 函数。

```
#define GetConst 1
```

当然，这种简单粗暴的做法即使有效，也会把 C++ 拉回“石器时代”。C++11 中对编译时期常量的回答是 constexpr，即常量表达式（constant expression）。比如我们要使得代码清单 6-1 中的 GetConst 函数成为一个常量表达式，可以用下面的声明方法：

```
constexpr int GetConst() { return 1; }
```

即在函数表达式前加上 constexpr 关键字即可。有了常量表达式这样的声明，编译器就可以在编译时期对 GetConst 表达式进行值计算（evaluation），从而将其视为一个编译时期的常量（虽然编译器不一定这么做，但至少从语法效果上看是这样，我们会在后面叙述）。这样一来代码清单 6-1 中的数组 Arr、匿名枚举的初始化以及 switch-case 的 case 表达式通过编译都不再是问题（读者可以自行实验一下）。

在 C++11 中，常量表达式实际上可以作用的实体不仅限于函数，还可以作用于数据声明，以及类的构造函数等。我们来分别看一下。

6.1.2 常量表达式函数

通常我们可以在函数返回类型前加入关键字 constexpr 来使其成为常量表达式函数。不过并非所有的函数都有资格成为常量表达式函数。事实上，常量表达式函数的要求非常严格，总结起来，大概有以下几点：

- 函数体只有单一的 return 返回语句。
- 函数必须返回值（不能是 void 函数）。
- 在使用前必须已有定义。
- return 返回语句表达式中不能使用非常量表达式的函数、全局数据，且必须是一个常量表达式。

让我们分别来解释一下。首先是常量表达式函数中最为明显的限制，就是要求函数体中只有一条语句，且该条语句必须是 return 语句。这就意味着形如：

```
constexpr int data() { const int i = 1; return i; }
```

这样的多条语句的写法是无法通过编译的。不过一些不会产生实际代码的语句在常量表达式函数中使用下，倒不会导致编译器的“抱怨”。我们可以看看如下 static_assert 的情况：

```
constexpr int f(int x){
    static_assert(0 == 0, "assert fail.");
    return x;
}
```

该例子能够通过编译。而其他的，比如 using 指令、typedef 等也通常不会造成问题。

第二点约束，则是常量表达式必须返回值。形如

```
constexpr void f(){}
```

这样的不返回值的函数就不能是常量表达式。当然，其原因也很明显，因为无法获得常量的常量表达式是不被认可的。

第三点约束是常量表达式函数在使用前必须被定义。对于普通函数而言，调用函数只需要有函数声明就够了，但常量表达式函数的使用则有所不同。这里读者应该注意常量表达式“使用”和“调用”的区别，前者讲的是编译时的值计算，而后者讲的是运行时的函数调用，如代码清单 6-3 所示。

代码清单 6-3

```
constexpr int f();
int a = f();
const int b = f();
constexpr int c = f();  // 无法通过编译
constexpr int f() { return 1; }
constexpr int d = f();

// 编译选项 :g++ -std=c++11 -c 6-1-3.cpp
```

这里我们声明了一个常量表达式函数 f。在定义该函数之前，我们定义了变量 a、常量 b 以及常量表达式值（下面即将介绍）c。在 a 和 b 的定义中，编译器会将 f() 转换为一个函数调用，而在 c 的定义中，由于其是一个常量表达式值，因此会要求编译器进行编译时的值计算。这时候由于 f 常量表达式还没有定义，就会导致编译错误。而 d 的定义则没有问题，因为 f 的定义已经有了。因此，在使用上读者必须小心。

其实从代码清单 6-3 我们也可以看出，虽然 f 被定义为一个常量表达式，但是它是否在编译时进行值计算则不一定。在 b 和 c 的定义中，f 就是标准的函数调用，constexpr 也不会使得函数被重写。那么普通情况下，程序员也就不用对声明了 constexpr 的函数再声明一个没有 constexpr 的版本了。而事实上，如果这么做还会导致错误发生，比如下面的声明就会导致编译器报错：

```
constexpr int f();
int f();
```

第四点非常重要，常量表达式中，也不能使用非常量表达式的函数。形如

```
const int e(){ return 1;}
constexpr int g(){ return e(); }
```

或者形如

```
int g = 3;
```

```
constexpr int h() { return g; }
```

的常量表达式定义是不能通过编译的。这样做的意义也比较明显，即如果我们要使得 g() 是一个编译时的常量，那么其 return 表达式语句就不能包含运行时才能确定返回值的函数。只有这样，编译器才能够在编译时进行常量表达式函数的值计算。

当然，作为一个常量表达式函数，return 的表达式需要是一个常量表达式也是天经地义的事情。一些危险的操作，比如赋值的操作在常量表达式中也是不允许的，形如

```
constexpr int k(int x) { return x = 1; }
```

的语句也是无法通过 C++11 编译器的编译的。

6.1.3 常量表达式值

在代码清单 6-3 中我们看到了由 constexpr 关键字修饰的“变量”c 和 f，这样声明的变量就是所谓的常量表达式值（constant-expression value）。通常情况下，常量表达式值必须被一个常量表达式赋值，而跟常量表达式函数一样，常量表达式值在使用前必须被初始化。

而使用 constexpr 声明的数据最常被问起的问题是，下列两条语句有什么区别：

```
const int i = 1;
constexpr int j = 1;
```

事实上，两者在大多数情况下是没有区别的。不过有一点是肯定的，就是如果 i 在全局名字空间中，编译器一定会为 i 产生数据。而对于 j，如果不是有代码显式地使用了它的地址，编译器可以选择不为它生成数据，而仅将其当做编译时期的值（是不是想起了光有名字没有产生数据的枚举值，以及不会产生数据的右值字面常量？事实上，它们也都只是编译时期的常量）。

这里还要提一下浮点常量。有的时候，我们在常量表达式中会看到浮点数。通常情况下，编译器对浮点数做编译时期常量这件事情很敏感。因为编译时环境和运行时环境可能有所不同，那么编译时的浮点常量和实际运行时的浮点数常量可能在精度上存在差别。不过在 C++11 中，编译时的浮点数常量表达式值还是被允许的。标准要求编译时的浮点常量表达式值的精度要至少等于（或者高于）运行时的浮点数常量的精度。

而对于自定义类型的数据，要使其成为常量表达式值的话，则不像内置类型这么简单。C++11 标准中，constexpr 关键字是不能用于修饰自定义类型的定义的。比如下面这样的类型定义和使用：

```
constexpr struct MyType {int i; }
constexpr MyType mt = {0};
```

在 C++11 中，就是无法通过编译的。正确地做法是，定义自定义常量构造函数（constent-

expression constructor)。我们可以看看代码清单 6-4 所示的例子。

代码清单 6-4

```
struct MyType {
    constexpr MyType(int x): i(x){}
    int i;
};
constexpr MyType mt = {0};

// 编译选项 :g++ -c -std=c++11 6-1-4.cpp
```

代码清单 6-4 中，我们对 MyType 的构造函数进行了定义。不过在定义前，我们加上了 constexpr 关键字。通过这样的定义，MyType 类型的 constexpr 的变量 mt 的定义就可以通过编译了。

常量表达式的构造函数也有使用上的约束，主要的有以下两点：

- 函数体必须为空。
- 初始化列表只能由常量表达式来赋值。

通常第二点跟常量表达式函数一样，是容易出错的，读者在使用中应该注意，形如下面的常量表达式构造函数都是无法通过编译的：

```
int f();
struct MyType { int i; constexpr MyType(): i(f()){}};
```

当然，这里还需要注意的是，虽然我们声明的是常量表达式构造函数，但是其编译时的“常量性”则体现在类型上，如代码清单 6-5 所示。

代码清单 6-5

```
#include <iostream>
using namespace std;

struct Date {
    constexpr Date(int y, int m, int d):
        year(y), month(m), day(d) {}

    constexpr int GetYear() { return year; }
    constexpr int GetMonth() { return month; }
    constexpr int GetDay() { return day; }

private:
    int year;
    int month;
    int day;
};
```

```
constexpr Date PRCfound {1949, 10, 1};
constexpr int foundmonth = PRCfound.GetMonth();

int main() { cout << foundmonth << endl; }  // 10

// 编译选项 :g++ -std=c++11 -c 6-1-5.cpp
```

在代码清单 6-5 中，我们为 Date 类型声明了常量表达式构造函数，随后定义了 constexpr 的变量 PRCfound。此外，还为 Date 定义常量表达式的成员函数，可以看到，可以从 PRCfound 中拿出成员 month，赋给一个常量表达式值 foundmonth。如果 PRCfound 的成员变量在这里不具有编译时的常量性，显然是不可能做到的。

这里还需要注意一下常量表达式的成员函数。在 C++11 中，不允许常量表达式作用于 virtual 的成员函数。这个原因也是显而易见的，virtual 表示的是运行时的行为，与“可以在编译时进行值计算”的 constexpr 的意义是冲突的。

跟常量表达式函数一样，常量表达式构造函数也可以用于非常量表达式中的类型构造，重写了编译器也会报错的。因而，程序员不必为类型再重写一个非常量表达式版本。

6.1.4 常量表达式的其他应用

常量表达式是可以用于模板函数的。不过由于模板中类型的不确定性，所以模板函数是否会被实例化为一个能够满足编译时常量性的版本通常也是未知的。针对这种情况，C++11 标准规定，当声明为常量表达式的模板函数后，而某个该模板函数的实例化结果不满足常量表达式的需求的话，constexpr 会被自动忽略。该实例化后的函数将成为一个普通函数，如代码清单 6-6 所示。

代码清单 6-6

```
struct NotLiteral{
    NotLiteral(){ i = 5; }
    int i;
};

NotLiteral nl;

template <typename T> constexpr T ConstExp(T t) {
      return t;
}

void g() {
    NotLiteral nl;
    NotLiteral nl1 = ConstExp(nl);
    constexpr NotLiteral nl2 = ConstExp(nl);     // 无法通过编译
```

```
    constexpr int a = ConstExp(1);
}

// 编译选项 :g++ -std=c++11 -c 6-1-6.cpp
```

在代码清单 6-6 中，结构体 NotLiteral 不是一个定义了常量表达式构造函数的类型，因此是不能够声明为常量表达式值的。而模板函数 ConstExp 一旦以 NotLiteral 为参数的话，那么其 constexpr 关键字将被忽略，如 nl1 变量所示。实例化为 ConstExp<NotLiteral> 的函数将不是一个常量表达式函数，因此，我们也看到 nl2 是无法通过编译的（当然，该例中 constexpr NotLiteral 本来也是不正确的声明）。而在可以实例化为常量表达式函数的时候，ConstExp 则可以用于常量表达式值的初始化。比如本例中的 a，就是由实例化为 ConstExp<int> 的常量表达式函数所初始化的。

对于常量表达式的应用，还有一个有趣的问题就是函数递归问题。如果一个函数是递归的，而且它是一个常量表达式函数，那么会发生什么情况呢？事实上，在常量表达式这个特性提出的时候，提出者是不太赞成让常量表达式支持递归的，不过 C++11 标准中并没有反对常量表达式的递归函数，而且在标准中说明，符合 C++11 标准的编译器对常量表达式函数应该至少支持 512 层的递归。

这就带来一些有趣的结果。有的时候，编译器被我们稍微改造一下，或许就能成为程序员的“计算器”。我们来看看代码清单 6-7 所示的例子。

代码清单 6-7

```
#include <iostream>
using namespace std;

constexpr int Fibonacci(int n) {
    return (n == 1) ? 1 : ((n == 2) ? 1 : Fibonacci(n - 1) + Fibonacci(n - 2));
}

int main() {
    int fib[] = {
        Fibonacci(11), Fibonacci(12),
        Fibonacci(13), Fibonacci(14),
        Fibonacci(15), Fibonacci(16)
    };

    for (int i : fib) cout << i << endl;
}

// 编译选项 :clang++ -std=c++11 6-1-7.cpp
```

这里程序员知道斐波那契数的算法，却懒得自行算出一个斐波那契数组（第 11~16 个），因此他利用了常量表达式构造了一个这样的数组。在我们的实验机上，我们用 clang++ 编译

器使用默认优化级别编译了这个程序，然后反汇编发现该数组的值已经被计算好了，实际运行的代码没有调用 Fibonacci 这个函数。这跟直接调用基于范围的 for 循环打印数组中的值的代码一致。

事实上，这种基于编译时期的运算的编程方式在 C++ 中并不是第一次出现。早在 C++ 模板刚出现的时候，就出现了基于模板的编译时期运算的编程方式，这种编程通常被称为模板元编程（template meta-programming）。模板元编程同样是非常有意思的话题，比如我们要实现代码清单 6-7 所示例子中的效果，使用模板元编程同样是可以的，如代码清单 6-8 所示。

代码清单 6-8

```
#include <iostream>
using namespace std;

template <long num>
struct Fibonacci{
    static const long val = Fibonacci<num - 1>::val + Fibonacci<num - 2>::val;
};

template <> struct Fibonacci<2> { static const long val = 1; };
template <> struct Fibonacci<1> { static const long val = 1; };
template <> struct Fibonacci<0> { static const long val = 0; };

int main() {
    int fib[] = {
        Fibonacci<11>::val, Fibonacci<12>::val,
        Fibonacci<13>::val, Fibonacci<14>::val,
        Fibonacci<15>::val, Fibonacci<16>::val,
    };

    for (int i : fib) cout << i << endl;
}

// 编译选项 :g++ -std=c++11 6-1-8.cpp
```

代码清单 6-8 中我们定义了一个非类型参数的模板 Fibonacci。该模板类定义了一个静态变量 val，而 val 的定义方式是递归的。因此模板将会递归地进行推导。此外，我们还通过偏特化定义了模板推导的边界条件，即斐波那契的初始值。那么模板在推导到边界条件的时候就会终止推导。通过这样的方法，我们同样可以在编译时进行值计算，从而生成数组的值。

通过 constexpr 进行的运行时值计算，跟模板元编程非常类似。因此有的程序员自然地称利用 constexpr 进行编译时期运算的编程方式为 constexpr 元编程（constexpr meta-programming）。学术地讲，constexpr 元编程与 template 元编程一样，都是图灵完备的，即任何程序中需要表达的计算，都可以通过 constexpr 元编程的方式来表达。由于 constexpr 支持浮点数运算（模板元编程只支持整型），支持三元表达式、逗号表达式，所以很多人认为

constexpr 元编程将会比模板元编程更加强大。从这个角度讲，constexpr 元编程将非常让人期待。

不过在这里我们还是必须为这些期待泼点冷水。因为从我们的实践中发现，并不是使用了 constexpr，编译器就一定会在编译时期对常量表达式函数进行值计算。事实上，对于代码清单 6-4 所示的例子，如果用 g++ 的默认优化级别来编译，我们实验机上会产生调用 Fibonacci 函数的代码（clang++ 在 O0 级别也会有这样的效果）。在 C++11 标准中，我们也没有看到要求编译器一定要对常量表达式进行编译时期的值计算。标准只是定义了可以用于编译时进行值运算的常量表达式的定义，却没有强制要求编译器一定在编译时进行值运算。所以编译器通过一些手段绕过代码的语法，仍然做运行时的调用，这样的方法也是符合 C++11 标准的（通常情况下，这样做也是编译器实现 constexpr 的第一步，因为这样最简单也最不容易出错，后期如果实现了编译时值计算，该方法还可以用作验证手段）。推迟到运行时的唯一的缺点就是性能上会有一定问题。可以想象，为了提高编译器的竞争力，各种编译器都会渐渐将常量表达式的运算放到编译时，到那个时候，constexpr 元编程或许才能大行其道。

6.2 变长模板

☞ 类别：库作者

6.2.1 变长函数和变长的模板参数

在 2.1 节中，我们知道 C++11 已经支持了 C99 的变长宏。变长宏与 printf 的默契配合使得程序员能够非常容易地派生出 printf 的变种以支持一些记录。而如我们提到的，printf 则使用了 C 语言的函数变长参数特性，通过使用变长函数（variadic funciton），printf 的实现能够接受任何长度的参数列表。不过无论是宏，还是变长参数，整个机制的设计上，没有任何一个对于传递参数的类型是了解的。我们可以看看变长函数的例子。通常情况下，一个变长函数可以如代码清单 6-9 所示。

代码清单 6-9

```
#include <stdio.h>
#include <stdarg.h>

double SumOfFloat(int count, ...) {
    va_list ap;
    double sum = 0;
    va_start(ap, count);              // 获得变长列表的句柄 ap
    for(int i = 0; i < count; i++)
        sum += va_arg(ap, double);    // 每次获得一个参数
    va_end(ap);
    return sum;
}
```

```
int main() {
    printf("%f\n", SumOfFloat(3, 1.2f, 3.4, 5.6));  // 10.200000
}

// 编译选项 :gcc 6-2-1.cpp
```

在代码清单 6-9 中，我们声明了一个名为 SumOfFloat 变长函数。变长函数的第一个参数 count 表示的是变长参数的个数，这必须由 SumOfFloat 的调用者传递进来。而在被调用者中，则需要通过一个类型为 va_list 的数据结构 ap 来辅助地获得参数。可以看到，这里代码首先使用 va_start 函数对 ap 进行初始化，使得 ap 成为被传递的变长参数的一个“句柄”（handler）。而后代码再使用 va_arg 函数从 ap 中将参数一一取出用于运算。由于这里是计算浮点数的和，所以每次总是给 va_arg 传递一个 double 类型作为参数。图 6-1 显示了一种变长函数的可能的实现方式，即以句柄 ap 为指向各个变长参数的指针，而 va_arg 则通过改变指针的方式（每次增加 sizeof(double) 字节）来返回下一个指针所指向的对象。

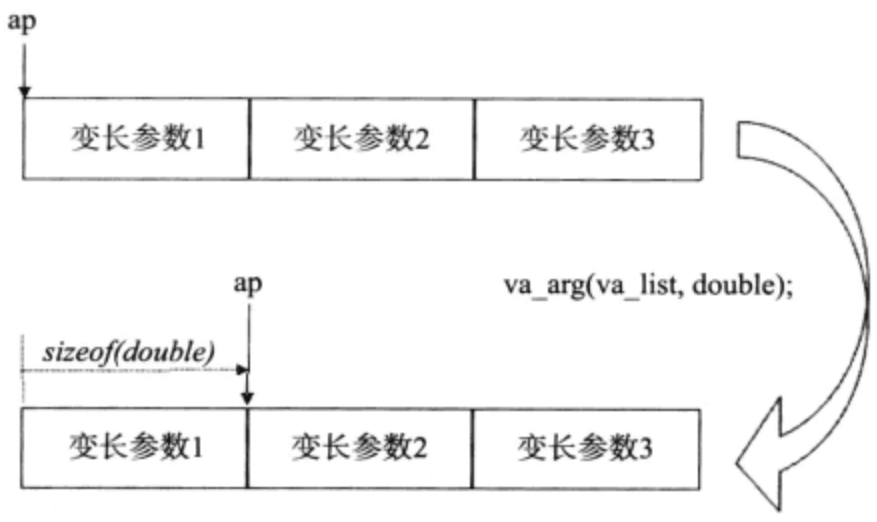


图 6-1 变长函数可能的实现方式

可以看到，在本例中，只有使用表达式 va_arg(ap, double) 的时候，我们才按照类型（实际是按类型长度）去变长参数列表中获得指定参数。而如何打印则得益于传递在字符串中的形如“%s”、“%d”这样的转义字，以及传递的 count 参数。事实上，函数“本身”完全无法知道参数数量或者参数类型。因此，对于一些没有定义转义字的非 POD 的数据来说，使用变长函数就会导致未定义的程序行为。比如：

```
const char *msg = "hello %s";
printf(msg, std::string("world"));
```

这样的代码就会导致 printf 出错。

从另一个角度讲，变长函数这种实现方式，对于 C++ 这种强调类型的语言来说相当于开了一个“不规范”的后门。这是 C++ 标准中所不愿意看到的（即使它能够工作）。因此，客

观上，C++ 需要引入一种更为“现代化”的变长参数的实现方式，即类型和变量同时能够传递给变长参数的函数。一个好的方式就是使用 C++ 的函数模板。在 C++98 中，标准要求函数模板始终具有数目确定的模板参数及函数参数。因此如果要实现一个新的变长参数的函数的话，我们需要突破参数的限制。

此外在一些情况下，类也需要不定长度的模板参数。最为典型的就是 C++11 标准库中的 tuple 类模板。如果读者熟悉 C++98 中的 pair 类模板的话，那么理解 tuple 也就不困难了。具体来讲，pair 是两个不同类型的数据的集合。比如 pair<int, double> 就能够容纳 int 类型和 double 类型的两种数据。一些如 std::map 的标准库容器，其成员就需要是类模板 pair 的。在 C++11 中，tuple 是 pair 类的一种更为泛化的表现形式。比起 pair，tuple 是可以接受任意多个不同类型的元素的集合。比如我们可以通过：

```
std::tuple<double, char, std::string> collections;
```

来声明一个 tuple 模板类。该 collections 变量可以容纳 double、char、std::string 三种类型的数据。当然，读者还可以用更多的参数来声明 collection，因为 tuple 可以接受任意多的参数。此外，和 pair 类似地，我们也可以更为简单地使用 C++11 的模板函数 make_tuple 来创造一个 tuple 模板类型。

```
std::make_tuple(9.8, 'g', "gravity");
```

由于 tuple 包含的类型数量可以任意地多，那么在客观上，就需要类模板能够接受变长的参数。因此，在 C++11 中我们就看到了所谓的变长模板（variadic template）的实现。

注意 在 C++98 中，由于没有变长模板，tuple 能够支持的模板参数数量实际上是有限的。这个数量是由标准库定义了多少个不同参数版本的 tuple 模板而决定的。

6.2.2 变长模板：模板参数包和函数参数包

我们先看看变长模板的语法，还是以前面提到的 tuple 为例，我们需要以下代码来声明 tuple 是一个变长类模板：

```
template <typename... Elements> class tuple;
```

可以看到，我们在标示符 Elements 之前的使用了省略号（三个“点”）来表示该参数是变长的。在 C++11 中，Elements 被称作是一个**“模板参数包”**（template parameter pack）。这是一种新的模板参数类型。有了这样的参数包，类模板 tuple 就可以接受任意多个参数作为模板参数。对于以下实例化的 tuple 模板类：

```
tuple<int, char, double>
```

编译器则可以将多个模板参数打包成为“单个的”模板参数包 Elements，即 Element 在进行模板推导的时候，就是一个包含 int、char 和 double 三种类型类型集合。

与普通的模板参数类似，模板参数包也可以是非类型的，比如：

```
template<int...A> class NonTypeVariadicTemplate{};
NonTypeVariadicTemplate<1, 0, 2> ntvt;
```

就定义了接受非类型参数的变长模板 NonTypeVariadicTemplate。这里，我们实例化一个三参数（1, 0, 2）的模板实例 ntvt。该声明方式相当于：

```
template<int, int, int> class NonTypeVariadicTemplate{};
NonTypeVariadicTemplate<1, 0, 2> ntvt;
```

这样的类模板定义和实例化。

除了类型的模板参数包和非类型的模板参数包，模板参数包实际上还是模板类型的，不过这样的声明会比较复杂，我们在后面再讨论。

一个模板参数包在模板推导时会被认为是模板的单个参数（虽然实际上它将会打包任意数量的实参）。为了使用模板参数包，我们总是需要将其**解包（unpack）**。在 C++11 中，这通常是通过一个名为**包扩展（pack expansion）**的表达式来完成。比如：

```
template<typename... A> class Template: private B<A...>{};
```

这里的表达式 A...（即参数包 A 加上三个“点”）就是一个包扩展。直观地看，参数包会在包扩展的位置展开为多个参数。比如：

```
template<typename T1, typename T2> class B{};
template<typename... A> class Template: private B<A...>{};
Template<X, Y> xy;
```

这里我们为类模板声明了一个参数包 A，而使用参数包 A... 则是在 Template 的私有基类 B<A...> 中，那么最后一个表达式就声明了一个基类为 B<X, Y> 的模板类 Template<X, Y> 的对象 xy。其中 X、Y 两个模板参数先是被打包为参数包 A，而后又在包扩展表达式 A... 中被还原。读者可以体会一下这样的使用方式。

不过上面对象 xy 的例子是基于类模板 B 总是接受两个参数的前提下的。倘若我们在这里声明了一个 Template<X, Y, Z>，就必然会发生模板推导的错误。这跟我们之前提到的“变长”似乎没有任何关系。那么如何才能利用模板参数包及包扩展，使得模板能够接受任意多的模板参数，且均能实例化出有效的对象呢？

事实上，在 C++11 中，实现 tuple 模板的方式给出了一种使用模板参数包的答案。这个思路是使用数学的归纳法，转换为计算机能够实现的手段则是递归。通过定义递归的模板偏特化定义，我们可以使得模板参数包在实例化时能够层层展开，直到参数包中的参数逐渐耗尽或到达某个数量的边界为止。下面的例子是一个用变长模板实现 tuple（简化的 tuple 实现）的代码，如代码清单 6-10 所示。

代码清单 6-10

```
template <typename... Elements> class tuple;     // 变长模板的声明

template<typename Head, typename... Tail>         // 递归的偏特化定义
class tuple<Head, Tail...> : private tuple<Tail...> {
    Head head;
};

template<> class tuple<> {};                                  // 边界条件

// 编译选项 :g++ -std=c++11 6-2-2.cpp
```

在代码清单 6-10 中，我们声明了变长模板类 tuple，其只包含一个模板参数，即 Elements 模板参数包。此外，我们又偏特化地定义了一个双参数的 tuple 的版本。该偏特化版本的 tuple 包含了两个参数，一个是类型模板参数 Head，另一个则是模板参数包 Tail。在代码清单 6-10 的实现中，我们将 Head 型的数据作为 tuple<Head, Tail...> 的第一个成员，而将使用了包扩展表达式的模板类 tuple<Tail...> 作为 tuple<Head, Tail...> 的私有基类。这样一来，当程序员实例化一个形如 tuple <double, int, char, float> 的类型时，则会引起基类的递归构造，这样的递归在 tuple 的参数包为 0 个的时候会结束。这是由于我们定义了边界条件或者说初始条件，即 tuple<> 这样不包含参数的偏特化版本而造成的。在代码清单 6-10 中，tuple<> 偏特化版本是一个没有成员的空类型。这样一来，编译器将从 tuple<> 建造出 tuple<float>，继而造出 tuple<char, float>、tuple<int, char, float>，最后就建造出了 tuple<double, int, char, float> 类型。

图 6-2 是 tuple <double, int, char, float> 实例化后的继承结构示意图。我们用方框表示类型，而方框内的方框则表示类型由其内部的方框所代表的类型私有派生而来。

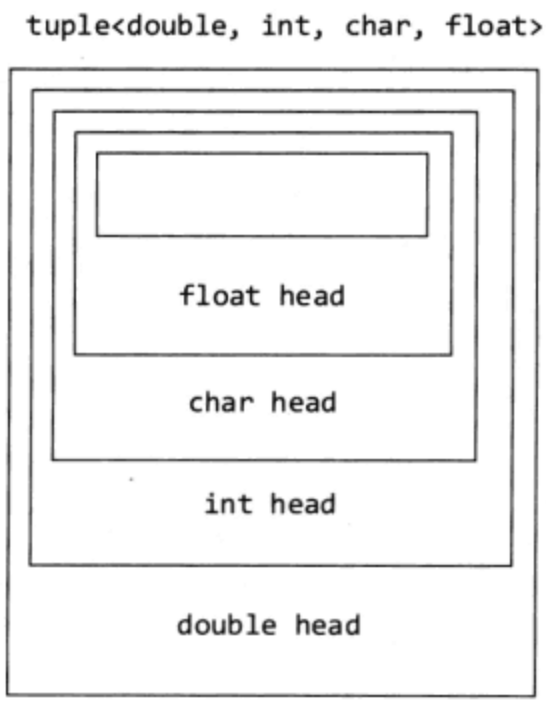


图 6-2　tuple<double, int, char, float> 继承结构示意图

这种变长模板的定义方式稍显复杂，不过却有效地解决了模板参数个数这样的问题。当然，这样做的前提是模板类 / 函数的定义要具有能够递推的结构。

我们再来看一个使用非类型模板的一个例子，如代码清单 6-11 所示。

代码清单 6-11

```
#include <iostream>
using namespace std;

template <long... nums> struct Multiply;

template <long first, long... last>
struct Multiply<first, last...> {
    static const long val = first * Multiply<last...>::val;
};

template<>
struct Multiply<> {
    static const long val = 1;
};

int main() {
    cout << Multiply<2, 3, 4, 5>::val << endl;          // 120
    cout << Multiply<22, 44, 66, 88, 9>::val << endl;   // 50599296
    return 0;
}

// 编译选项 :clang++ -std=c++11 6-2-3.cpp
```

在代码清单 6-11 中，我们定义了接受非类型参数的变长模板类 Multiply。Multiply 的唯一用途是将模板的参数相乘。而通过 Mutiply 的成员 val，我们可以将参数相乘的结果读出来。类似地，这里也发生了编译时期的值计算，我们最后在 main 函数中打印的是 Multiply 的一个常量静态成员 val，而非执行任何函数调用。本例跟代码清单 6-8 的例子非常相似，也可以算作模板元编程的范畴。

除了变长的模板类，在 C++11 中，我们还可以声明变长模板的函数。对于变长模板函数而言，除了声明可以容纳变长个模板参数的模板参数包之外，相应地，变长的函数参数也可以声明成函数参数包（function parameter pack）。比如：

```
template<typename ... T> void f(T ... args);
```

这个例子中，由于 T 是个变长模板参数（类型），因此 args 则是对应于这些变长类型的数据，即函数参数包。值得注意的是，在 C++11 中，标准要求函数参数包必须唯一，且是函数的最后一个参数（模板参数包没有这样的要求）。

有了模板参数包和函数参数包两个概念，我们就可以实现 C 中变长函数的功能了。我们

可以看看这个 C++11 提案中实现新的 printf 的例子，如代码清单 6-12 所示。

代码清单 6-12

```
#include <iostream>
#include <stdexcept>
using namespace std;

void Printf(const char* s) {
    while (*s) {
        if (*s == '%' && *++s != '%')
            throw runtime_error("invalid format string: missing arguments");
        cout << *s++;
    }
}

template<typename T, typename... Args>
void Printf(const char* s, T value, Args... args) {
    while (*s) {
        if (*s == '%' && *++s != '%') {
            cout << value;
            return Printf(++s, args...);
        }
        cout << *s++;
    }
    throw runtime_error("extra arguments provided to Printf");
}

int main() {
    Printf("hello %s\n", string("world"));  // hello world
}

// 编译选项 :g++ -std=c++11 6-2-4.cpp
```

在代码清单 6-12 中，实现了 Printf 的参数。Printf 是一个变长函数模板，这里为了兼容于 printf，仍然提供了 printf 式的字符串，所以 Printf 功能等同于 printf，可以接受该字符串及变长的参数。而 Printf 的功能也大于 printf，因为它可以接受 std::string 这样的非内置类型。

从代码清单 6-12 的代码中我们看到，相比于变长函数，变长函数模板不会丢弃参数的类型信息。因此重载的 cout 的操作符 << 总是可以将具有类型的变量正确地打印出来。这就是 Printf 功能大于 printf 的主要原因，也是变长模板函数远强于变长函数的地方。

6.2.3 变长模板：进阶

在代码清单 6-10 中，我们看见参数包在类的基类描述列表中进行了展开，而在代码清单 6-11 中，参数包则在表达式中进行了展开。那么在 C++11 中，有多少地方可以展开参数包呢？事实上，标准定义了以下 7 种参数包可以展开的位置：

- ❑表达式
- ❑初始化列表（列表初始化参见第 3 章）
- ❑基类描述列表
- ❑类成员初始化列表
- ❑模板参数列表
- ❑通用属性列表（第 8 章中会讲到）
- ❑lambda 函数的捕捉列表（第 7 章中会讲到）

语言的其他“地方”则无法展开参数包。而对于包扩展而言，其解包也与其声明的形式息息相关。事实上，我们还可以声明一些有趣的包扩展表达式。比如声明了 Arg 为参数包，那么我们可以使用 Arg&&... 这样的包扩展表达式，其解包后等价于 Arg1&&,...,Argn&&（这里我们假设 Arg1 是参数包里的第一个参数，而 Argn 是最后一个）。

一个更为有趣的包扩展表达式如下：

```
template<typename... A> class T: private B<A>...{};
```

注意这个包扩展跟下面的类模板声明：

```
template<typename... A> class T: private B<A...>{};
```

在解包后是不同的，对于同样的实例化 T<X, Y>，前者会解包为：

```
class T<X, Y>: private B<X>, private B<Y>{};
```

即多重继承的派生类，而后者则会解包为：

```
class T<X, Y>: private B<X, Y>{};
```

即派生于多参数的模板类的派生类，这点存在着本质的不同。

类似的状况也会发生在函数声明上，我们可以看看代码清单 6-13 所示的例子。

代码清单 6-13

```
#include <iostream>
using namespace std;

template<typename ... T> void DummyWrapper(T... t) {}

template <typename T> T pr(T t) {
    cout << t;
    return t;
}

template<typename... A>
void VTPrint(A... a) {
    DummyWrapper(pr(a)...); // 包扩展解包为 pr(1), pr(", ")..., pr(", abc\n")
};
```

```
int main() {
    VTPrint(1, ", ", 1.2, ", abc\n");
}

// 编译选项：xlC -+ -qlanglvl=variadictemplates 6-2-5.cpp
```

在代码清单 6-13 中，我们定义了一个依赖于变长模板包扩展展开的 VTPrint 函数。可以看到，pr(a)... 这样的包扩展表达式，将会被展开为 pr(1)、pr(", ")、...、pr(", abc\n")，从而 pr 将参数以此打印出来。在我们的实验机上，我们可以得到以下输出：

```
1, 1.2, abc
```

这样一来，我们也实现一个接受任意长度参数的打印函数。可以看到，这样的包扩展表达式，为参数包的使用带来了非常多的灵活性。

注意 在我们实验机上使用 g++ 编译上述例子将无法得到上面的答案（g++ 似乎是被每个 pr 的执行顺序打乱了）。

以上讲的都是参数包的扩展，事实上，除去扩展外，在 C++11 中，标准还引入了新操作符“sizeof...”（没有看错，这个操作符就是 sizeof 后面加上了 3 个小点），其作用是计算参数包中的参数个数。通过这个操作符，我们能够实现参数包更多的用法。我们来看一个例子，如代码清单 6-14 所示。

代码清单 6-14

```
#include <cassert>
#include <iostream>
using namespace std;

template<class...A> void Print(A...arg) {
    assert(false);  // 非 6 参数偏特化版本都会默认 assert(false)
}

// 特化 6 参数的版本
void Print(int a1, int a2, int a3, int a4, int a5, int a6) {
    cout << a1 << ", " << a2 << ", " << a3 << ", "
        << a4 << ", " << a5 << ", " << a6 << endl;
}

template<class...A> int Vaargs(A...args){
    int size = sizeof...(A);    // 计算变长包的长度

    switch(size){
        case 0: Print(99, 99, 99, 99, 99, 99);
                break;
        case 1: Print(99, 99, args..., 99, 99, 99);
```

```
                break;
        case 2: Print(99, 99, args..., 99, 99);
                break;
        case 3: Print(args..., 99, 99, 99);
                break;
        case 4: Print(99, args..., 99);
                break;
        case 5: Print(99, args...);
                break;
        case 6: Print(args...);
                break;
        default:
                Print(0, 0, 0, 0, 0, 0);
    }
    return size;
}

int main(void){
    Vaargs();   // 99, 99, 99, 99, 99, 99
    Vaargs(1);  // 99, 99, 1, 99, 99, 99
    Vaargs(1, 2);   // 99, 99, 1, 2, 99, 99
    Vaargs(1, 2, 3);    // 1, 2, 3, 99, 99, 99
    Vaargs(1, 2, 3, 4); // 99, 1, 2, 3, 4, 99
    Vaargs(1, 2, 3, 4, 5);  // 99, 1, 2, 3, 4, 5
    Vaargs(1, 2, 3, 4, 5, 6);   // 1, 2, 3, 4, 5, 6
    Vaargs(1, 2, 3, 4, 5, 6, 7);    // 0, 0, 0, 0, 0, 0
    return 0;
}

// 编译选项 :g++ -std=c++11 6-2-6.cpp
```

在代码清单 6-14 的例子当中，我们仅为变长函数模板 Print 提供了一个参数为 6 的特化版本。假如我们的代码试图实例化参数不等于 6 的 Print，编译器则会推导 Print 为执行 assert(false) 的操作的版本。反之，则会实例化参数为 6 的特化版本。这里，Vaargs 的函数参数包的参数数量被计算并被传递，进而实现不同参数数量的不同打印。这里同样实现了接受任意个参数的功能，不过却没有像之前一样使用递归。

有的读者一定会对使用模板做变长模板的参数包感兴趣。实际上，使用模板做参数包跟使用类型和非类型的模板参数包并没有太大的区别。我们可以看看代码清单 6-15 所示的这个例子。

代码清单 6-15

```
template<typename I, template<typename> class... B> struct Container{};

template<typename I, template<typename> class A, template<typename> class... B>
struct Container<I, A, B...> {
```

```
    A<I> a;
    Container<I, B...> b;
};

template<typename I> struct Container<I>{};

// 编译选项 :g++ -std=c++11 -c 6-2-7.cpp
```

代码清单 6-15 就定义了使用模板作为变长模板参数包的一个变长模板。可以看到，Container 类型使用了两个参数——类型 I 和模板参数包 template<typename> class...B。而递归的偏特化定义中，我们看到成员变量的定义使用了类型 I 和模板 A：A<I> 这样的模板中的类型声明。即用 I 做参数实例化模板参数 template<typename> class A（希望读者还没有糊涂）。而 Container<I, B...> b 则保证编译器会继续递归地推导下去。推导到模板参数包为空的时候，我们的边界条件就会起作用。代码清单 6-15 整段代码并没有特别之处，如果读者在这里觉得阅读代码有点儿困难，那可能需要花一点儿时间编写代码来进行理解。

那么，在 C++11 中，我们是否可以同时在变长模板类中声明两个模板参数包呢？事实上，如果我们像下面这样声明，通常就会发生编译错误：

```
template<class...A, class...B> struct Container{};
template<class...A, class B> struct Container{};
```

编译器会提示程序员，模板参数包不是变长模板类的最后一个参数。不过实际上，如果编译器能够进行推导的话，模板参数包倒不一定非得作为模板的最后一个参数，比如代码清单 6-16 所示的这个例子[⊖]。

代码清单 6-16

```
#include <cstdio>
#include <tuple>
using namespace std;

template<typename A, typename B> struct S {};

// 两个模板参数包
template<
    template<typename... > class T, typename... TArgs
  , template<typename... > class U, typename... UArgs
  >
  struct S< T< TArgs... >, U< UArgs... > > {};

int main()
{
    S<int, float>  p;
```

⊖ 本例来源于 http://stackoverflow.com/questions/4706677/partial-template-specialization-with-multiple-template-parameter-packs。

```
    S<std::tuple<int, char>, std::tuple<float>>  s;
}

// 编译选项 :g++ -std=c++11 6-2-8.cpp
```

代码清单 6-16 中，我们使用了两个模板参数包作为模板类 S 的参数。很有意思的是，除了 struct S 以外，另外两个模板参数：class T 和 class U 也是变长模板，因此本例实际是一个变长模板作参数的模板示例。值得注意的是，模板中的变长模板是无法做递归的偏特化声明和定义边界条件的特化声明的。不过在本例中，我们看到了模板类 struct S 依然是可以推导的。这是因为我们使用了标准库中的变长模板类型 tuple。那么 T 和 U 的推导就可以根据 tuple 的定义进行，直至推导到边界条件（T 和 U 两个 tuple 都不再包含模板参数），即 template<typename A, typename B> struct S {};，S 的定义才递归地被产生。

代码清单 6-16 的写法略有些晦涩，不过至少成功地同时地使用两个模板参数包。如果读者需要更多的模板参数包，也可以“依葫芦画瓢”。

我们再来看看变长模板参数和完美转发结合使用的例子，如代码清单 6-17 所示。

代码清单 6-17

```
#include <iostream>
using namespace std;

struct A {
    A(){}
    A(const A& a) { cout << "Copy Constructed " << __func__ << endl; }
    A(A&& a) { cout << "Move Constructed " << __func__ << endl; }
};

struct B {
    B(){}
    B(const B& b) { cout << "Copy Constructed " << __func__ << endl; }
    B(B&& b) { cout << "Move Constructed " << __func__ << endl; }
};

// 变长模板的定义
template <typename... T> struct MultiTypes;
template <typename T1, typename... T>
struct MultiTypes<T1, T...> : public MultiTypes<T...>{
    T1 t1;
    MultiTypes<T1, T...>(T1 a, T... b):
        t1(a), MultiTypes<T...>(b...) {
            cout << "MultiTypes<T1, T...>(T1 a, T... b)" << endl;
        }
};
template <> struct MultiTypes<> {
    MultiTypes<>(){ cout << "MultiTypes<>()" << endl;}
```

```
};

// 完美转发的变长模板
template <template <typename...> class VariadicType, typename... Args>
VariadicType<Args...> Build(Args&&... args)
{
    return VariadicType<Args...>(std::forward<Args>(args)...);
}

int main() {
    A a;
    B b;

    Build<MultiTypes>(a, b);
}

// 编译选项 :g++ -std=c++11 6-2-9.cpp
```

在代码清单 6-17 中，我们定义了两个类型 A 和 B。此外，我们还定义了变长模板 MultiType，以及一个接受变长模板作为参数的变长模板函数 Build。可以看到，在 Build 的声明中，我们将其参数声明为一个右值引用，而我们在转发时，则使用了 std::forward 来保证左值按照左值引用、右值按照右值引用的方式传递。在我们的这段代码示例中，main 函数传递了两个左值给 Build<MultiTypes> 作为变长函数包。这里，我们通过 Multitypes 的构造函数来递归地构造一个 MultiTypes 的实例。编译运行该程序，在我们的实验机上，可以看到如下结果：

```
MultiTypes<>()
MultiTypes<T1, T...>(T1 a, T... b)
MultiTypes<T1, T...>(T1 a, T... b)
```

虽然代码清单 6-17 所示的代码看起来非常复杂，不过其产生的效果却非常好。事实上，由于我们传递的是左值，因此在 Multitypes 对象在构造的时候，没有调用任何的拷贝构造函数或者移动构造函数。构造后的类型实际上只包含了对之前定义变量 a 和 b 的引用。我们可以通过图 6-3 来看一下。

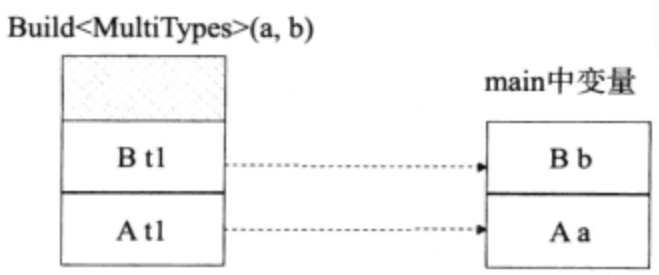


图 6-3　代码清单 6-17 中构造的变量类型

虽然在语法和编程上，使用变长模板会存在一些难度，不过对于库的编写者而言，变长模板具备了很好的实用性（尤其是它能够实现其他方式无法实现的功能）。在标准库中，也

添加了形如 tuple、emplace_back 这样的变长模板类和变长模板函数。如果读者需要传递变长的类型或者函数参数，也不妨使用变长模板试试。

6.3 原子类型与原子操作

☞ 类别：所有人

6.3.1 并行编程、多线程与 C++11

在 C++11 之前，C/C++ 一直是一种顺序的编程语言。顺序是指所有指令都是串行执行的，即在相同的时刻，有且仅有单个 CPU 的程序计数器指向可执行代码的代码段，并运行代码段中的指令。而 C/C++ 代码也总是对应地拥有一份操作系统赋予进程的包括堆、栈、可执行的（代码）及不可执行的（数据）在内的各种内存区域。

不过随着处理器的发展，半导体工业在提升处理器频率时遭遇到漏电流等各种技术瓶颈。以顺序执行编程模型为基础的单核处理器的发展，在高速发展 20 多年后开始接近停滞。随之而来的是多核处理器的发展风潮。相应地，编程语言逐渐也开始向并行化的编程方式发展。

常见的并行编程有多种模型，如共享内存、多线程、消息传递等。不过从实用性上讲，多线程模型往往具有较大的优势。多线程模型允许同一时间有多个处理器单元执行统一进程中的代码部分，而通过分离的栈空间和共享的数据区及堆栈空间，线程可以拥有独立的执行状态以及进行快速的数据共享。因此在 2000 年以后，主流的芯片厂商以及编译器开发厂商或组织都开始推广适用于多核处理器的多线程编程模型。而编程语言，也逐渐地将线程模型纳入语言特性或者语言库中。不过 C/C++ 由于新标准迟迟未出，因此我们也就一直没有看到集成于 C/C++ 语言特性中的线程特性或者线程库。

在 C++11 之前，在 C/C++ 中程序中使用线程却并非鲜见。这样的代码主要使用 POSIX 线程（Pthread）和 OpenMP 编译器指令两种编程模型来完成程序的线程化。其中，POSIX 线程是 POSIX 标准中关于线程的部分，程序员可以通过一些 Pthread 线程的 API 来完成线程的创建、数据的共享、同步等功能。Pthread 主要用于 C 语言，在类 UNIX 系统上，如 FreeBSD、NetBSD、OpenBSD、GNU/Linux、Mac OS X，甚至是 Windows 上都有实现（Windows 上 Pthread 的实现并非“原生”，主要还是包装为 Windows 的线程库）。不过在使用的便利性上，Pthread 不如后来者 OpenMP。OpenMP 的编译器指令将大部分的线程化的工作交给了编译器完成，而将识别需要线程化的区域的工作交给了程序员，这样的使用方式非常简单，也易于推广。因此，OpenMP 得到了业界大多数主流软硬件厂商，如 AMD、IBM、Intel、Cray、HP、Fujitsu、Nvidia、NEC、Microsoft、Texas Instruments、Oracle Corporation 等的支持。除去 C/C++ 语言外，OpenMP 还可以用于 Fortran 语言，是现行的一种非常有影响力的使用线程程序优化的编程模型。

而在 C++11 中，标准的一个相当大的变化就是引入了多线程的支持。这使得 C/C++ 语言在进行线程编程时，不必依赖第三方库和标准。而 C/C++ 对线程的支持，一个最为重要的部分，就是在原子操作中引入了原子类型的概念。

6.3.2 原子操作与 C++11 原子类型

所谓原子操作，就是多线程程序中“最小的且不可并行化的”的操作。通常对一个共享资源的操作是原子操作的话，意味着多个线程访问该资源时，有且仅有唯一一个线程在对这个资源进行操作。那么从线程（处理器）的角度看来，其他线程就不能够在本线程对资源访问期间对该资源进行操作，因此原子操作对于多个线程而言，就不会发生有别于单线程程序的意外状况。

通常情况下，原子操作都是通过“互斥”（mutual exclusive）的访问来保证的。实现互斥通常需要平台相关的特殊指令，这在 C++11 标准之前，这常常意味着需要在 C/C++ 代码中嵌入内联汇编代码。对程序员来讲，就必须了解平台上与同步相关的汇编指令。当然，如果只是想实现粗粒度的互斥，借助 POSIX 标准的 pthread 库中的互斥锁（mutex）也可以做到。我们可以看看代码清单 6-18 所示的例子。

代码清单 6-18

```
#include <pthread.h>
#include <iostream>
using namespace std;

static long long total = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

void* func(void *) {
    long long i;
    for(i = 0; i < 100000000LL;i++) {
        pthread_mutex_lock(&m);
        total += i;
        pthread_mutex_unlock(&m);
    }
}

int main() {
    pthread_t thread1, thread2;
    if (pthread_create(&thread1, NULL, &func, NULL)){
        throw;
    }
    if (pthread_create(&thread2, NULL, &func, NULL)){
        throw;
    }
    pthread_join(thread1, NULL);
```

```
    pthread_join(thread2, NULL);
    cout << total << endl;  // 9999999900000000
    return 0;
}

// 编译选项 :g++ 6-3-1.cpp -lpthread
```

在代码清单 6-18 中，我们给出了一个在 Linux 上使用 pthread 进行原子操作的例子。这里，为了保证 total += i 语句的原子性，我们创建了一个 pthread_mutex_t 类型的互斥锁 m，并且在语句的前后使用加锁（pthread_mutex_lock）和解锁（pthread_mutex_unlock）两种操作来确保该语句只有单一线程可以访问。这里我们启动了两个线程 thread1 和 thread2，并将它们加入（join）程序的执行。由于两个线程互斥地访问原子操作语句，从而得出 total 正确结果为 9999999900000000。对于多线程的程序而言，进出临界区（即我们的原子操作语句 total += i）的加锁 / 解锁操作都是必须的。如果将加锁 / 解锁操作的代码都注释掉的话，在我们的实验机上，total 的结果将由于线程间对数据的竞争（contention）而不再准确。因此，为了防止数据竞争问题，我们总是需要确保对 total 的操作是原子操作。

不过显而易见地，代码清单 6-18 中基于 pthread 的方法虽然可行，但代码编写却很麻烦。程序员需要为共享变量创建互斥锁，并在进入临界区前后进行加锁和解锁的操作。对于习惯了在单线程情况下编程的程序员而言，互斥锁的管理无疑是种负担。不过在 C++11 中，通过对并行编程更为良好的抽象，要实现同样的功能就简单了很多。我们可以看看代码清单 6-19 所示的例子。

代码清单 6-19

```
#include <atomic>
#include <thread>
#include <iostream>
using namespace std;

atomic_llong total {0};      // 原子数据类型

void func(int) {
    for(long long i = 0; i < 100000000LL; ++i) {
        total += i;
    }
}

int main() {
    thread t1(func, 0);
    thread t2(func, 0);

    t1.join();
    t2.join();
    cout << total << endl;  // 9999999900000000
    return 0;
```

```
}

// 编译选项 :g++ -std=c++11 6-3-2.cpp -lpthread
```

在代码清单 6-19 中，我们将变量 total 定义为一个“原子数据类型”：atomic_llong，该类型长度等同于 C++11 中的内置类型 long long。在 C++11 中，程序员不需要为原子数据类型显式地声明互斥锁或调用加锁、解锁的 API，线程就能够对变量 total 互斥地进行访问。这里我们定义了 C++11 的线程 std::thread 变量 t1 及 t2，它们都执行同样的函数 func，并类似于 pthread_t，调用了 std::thread 成员函数 join 加入程序的执行。可以看到，由于原子数据类型的原子性得到了可靠的保障，程序最后打印出的 total 的值依然为 9999999900000000。

相比于基于 C 以及过程编程的 pthread“原子操作 API”而言，C++11 对于“原子操作”概念的抽象遵从了面向对象的思想——C++11 标准定义的都是所谓的“原子类型”。而传统意义上所谓的“原子操作”，则抽象为针对于这些原子类型的操作（事实上，是原子类型的成员函数，稍后解释）。直观地看，编译器可以保证原子类型在线程间被互斥地访问。这样设计，从并行编程的角度看，是由于需要同步的总是数据而不是代码，因此 C++11 对数据进行抽象，会有利于产生行为更为良好的并行代码。而进一步地，一些琐碎的概念，比如互斥锁、临界区则可以被 C++11 的抽象所掩盖，因而并行代码的编写也会变得更加简单。

在 C++11 的并行程序中，使用原子类型是非常容易的。事实上，由于 C++11 与 C11 标准都支持原子类型，因此我们可以简单地通过 #include <cstdatomic> 头文件中来使用对应于内置类型的原子类型定义。<cstdatomic> 中包含的原子类型定义如表 6-1 所示。

表 6-1 <cstdatomic> 中原子类型和内置类型对应表

原子类型名称	对应的内置类型名称
atomic_bool	bool
atomic_char	char
atomic_schar	signed char
atomic_uchar	unsigned_char
atomic_int	int
atomic_uint	unsigned int
atomic_short	short
atomic_ushort	unsigned short
atomic_long	long
atomic_ulong	unsigned long
atomic_llong	long long
atomic_ullong	unsigned long long
atomic_char16_t	char16_t
atomic_char32_t	char32_t
atomic_wchar_t	wchar_t

代码清单 6-19 就采用了这样的方式。不过更为普遍地，程序员可以使用 atomic 类模板。通过该类模板，程序员任意定义出需要的原子类型。比如下列语句：

```
std::atomic<T> t;
```

就声明了一个类型为 T 的原子类型变量 t。编译器会保证产生并行情况下行为良好的代码，以避免线程间对数据 t 的竞争。而在 C11 中，要想定义原子的自定义类型，则需要使用 C11 的新关键字 _Atomic 来完成（不过在本书完成时，各个编译器对 C11 中原子操作的支持都非常有限）。

对于线程而言，原子类型通常属于“资源型”的数据，这意味着多个线程通常只能访问单个原子类型的拷贝。因此在 C++11 中，原子类型只能从其模板参数类型中进行构造，标准不允许原子类型进行拷贝构造、移动构造，以及使用 operator= 等，以防止发生意外。比如：

```
atomic<float> af {1.2f};
atomic<float> af1 {af};  // 无法通过编译
```

其中，af1{af} 的构造方式在 C++11 中是不允许的（事实上，atomic 模板类的拷贝构造函数、移动构造函数、operator= 等总是默认被删除的。我们会在第 7 章中介绍如何删除一些默认的函数）。

不过从 atomic<T> 类型的变量来构造其模板参数类型 T 的变量则是可以的。比如：

```
atomic<float> af {1.2f};
float f = af;
float f1 {af};
```

这是由于 atomic 类模板总是定义了从 atomic<T> 到 T 的类型转换函数的缘故。在需要时，编译器会隐式地完成原子类型到其对应的类型的转换。

那么，使得原子类型能够在线程间保持原子性的缘由主要还是因为编译器能够保证针对原子类型的操作都是原子操作。如我们之前提到的，原子操作都是平台相关的，因此有必要为常见的原子操作进行抽象，定义出统一的接口，并根据编译选项（或环境）产生其平台相关的实现。在 C++11 中，标准将原子操作定义为 atomic 模板类的成员函数，这囊括了绝大多数典型的操作，如读、写、交换等。当然，对于内置类型而言，主要是通过重载一些全局操作符来完成的。以之前在代码清单 6-19 中看到的 operator+= 操作为例，在我们的实验机上使用 g++ 进行编译的话，会产生一条特殊的 lock 前缀的 x86 指令，lock 能够控制总线及实现 x86 平台上的原子性的加法。

表 6-2 显示了所有 atomic 类型及其相关的操作。

表 6-2 atomic 类型的操作

操作	atomic_flag	atomic_bool	atomic-integral-type	atomic <bool>	atomic <T*>	atomic <integral-type>	Atomic <class-type>
test_and_set	Y						
clear	Y						
is_lock_free		y	y	y	y	y	y
load		y	y	y	y	y	y
store		y	y	y	y	y	y
exchange		y	y	y	y	y	y
compare_exchange_weak +strong		y	y	y	y	y	y
fetch_add, +=			y		y	y	
fetch_sub, -=			y		y	y	
fetch_or, \|=		y			y		
fetch_and, &=			y			y	
fetch_xor, ^=			y			y	
++,--			y		y	y	y

这里的 atomic-integral-type 和 integral-type，指的都是表 6-1 中所有原子类型的整型，而 class-type 则是指自定义类型。可以看到，对于大多数的原子类型而言，都可以执行读（load）、写（store）、交换（exchange）、比较并交换（compare_exchange_weak/compare_exchange_stronge）等操作。通常情况下，这些原子操作已经足够使用了。比如在下列语句中：

```
atomic<int> a;
int b = a;
```

赋值语句 b = a 实际就等同于 b = a.load()。而由于 a.load 是原子操作，因此可以避免线程间关于 a 的竞争，而下列语句：

```
atomic<int> a;
a = 1;
```

其赋值语句 a = 1 则等同于调用 a.store(1)。同样的，由于 a.store 是原子操作，也可以避免线程间关于 a 的竞争。而 exchange 和 compare_exchange_weak/compare_exchange_stronge 则更为复杂一些。由于每个平台上对线程间实现交换、比较并交换等操作往往有着不同的方式，无法用一致的高级语言表达，因此这些接口封装了平台上最高性能的实现，使得程序员能够在不同平台上都能获得最佳的性能。

此外，对于整型和指针类型，我们还可以看到，标准为其定义了一些算术运算和逻辑运算的操作符，其意义都比较直观，就不赘述了。不过在表 6-2 中，我们还可以看到一个比较特殊的布尔型的 atomic 类型：atomic_flag（注意，atomic_flag 跟 atomic_bool 是不同的），相比于其他的 atomic 类型，atomic_flag 是无锁的（lock-free），即线程对其访问不需要加锁，因

此对 atomic_flag 而言，也就不需要使用 load、store 等成员函数进行读写（或者重载操作符）。而典型地，通过 atomic_flag 的成员 test_and_set 以及 clear，我们可以实现一个自旋锁（spin lock）。我们来看看代码清单 6-20 所示的例子。

代码清单 6-20

```
#include <thread>
#include <atomic>
#include <iostream>
#include <unistd.h>
using namespace std;

std::atomic_flag lock = ATOMIC_FLAG_INIT;

void f(int n) {
    while (lock.test_and_set(std::memory_order_acquire)) // 尝试获得锁
        cout << "Waiting from thread " << n << endl;      // 自旋
    cout << "Thread " << n << " starts working" << endl;
}

void g(int n) {
    cout << "Thread " << n << " is going to start." << endl;
    lock.clear();
    cout << "Thread " << n << " starts working" << endl;
}

int main() {
    lock.test_and_set();
    thread t1(f, 1);
    thread t2(g, 2);

    t1.join();
    usleep(100);
    t2.join();
}

// 编译选项 :g++ -std=c++11 6-3-3.cpp -lpthread
```

在代码清单 6-20 中，我们声明了一个全局的 atomic_flag 变量 lock。最开始，将 lock 初始化为值 ATOMIC_FLAG_INIT，即 false 的状态。而在线程 t1 中（执行函数 f 的代码），我们不停地通过 lock 的成员 test_and_set 来设置 lock 为 true。这里的 test_and_set() 是一种原子操作，用于在一个内存空间原子地写入新值并且返回旧值。因此 test_and_set 会返回之前的 lock 的值。所以当线程 t1 执行 join 之后，由于在 main 函数中调用过 test_and_set，因此 f 中的 test_and_set 将一直返回 true，并不断打印信息，即自旋等待。而当线程 t2 加入运行的时候，由于其调用了 lock 的成员 clear，将 lock 的值设为 false，因此此时线程 t1 的自旋将终止，从而开始运行后面的代码。这样一来，我们实际上就通过自旋锁达到了让 t1 线程等待 t2

线程的效果。当然，还可以将 lock 封装为锁操作，比如：

```
void Lock(atomic_flag *lock) { while (lock.test_and_set ()); }
void Unlock(atomic_flag *lock) { lock.clear(); }
```

这样一来，就可以通过 Lock 和 UnLock 操作，像“往常”一样互斥地访问临界区了。除此之外，很多时候，了解底层的程序员会考虑使用无锁编程，以最大限度地挖掘并行编程的性能，而 C++11 的无锁机制为这样的实现提供了高级语言的支持。

在上面的例子中，我们的原子操作都是比较直观的。事实上，在 C++11 中，原子操作还可以包含一个参数：memory_order。通常情况下，使用该参数将有利于编译器进一步释放并行的潜在的性能。不过在这之前，我们必须先了解一下什么是内存模型。

6.3.3 内存模型，顺序一致性与 memory_order

如果只是简单地想在线程间进行数据的同步的话，原子类型已经为程序员已经提供了一些同步的保障。不过这样做的安全性却是建筑于一个假设之上，即所谓的顺序一致性（sequential consistent）的内存模型（memory model）。要了解顺序一致性以及内存模型，我们不妨看看代码清单 6-21 所示的例子。

代码清单 6-21

```
#include <thread>
#include <atomic>
#include <iostream>
using namespace std;

atomic<int> a {0};
atomic<int> b {0};

int ValueSet(int) {
    int t = 1;
    a = t;
    b = 2;
}

int Observer(int) {
    cout << "(" << a << ", " << b << ")" << endl;    // 可能有多种输出
}

int main() {
    thread t1(ValueSet, 0);
    thread t2(Observer, 0);

    t1.join();
    t2.join();
    cout << "Got (" << a << ", " << b << ")" << endl;    // Got (1, 2)
```

```
}
// 编译选项 :g++ -std=c++11 6-3-4.cpp -lpthread
```

在代码清单 6-21 中，我们创建了两个线程 t1 和 t2，分别执行 ValueSet 和 Observer 函数。在 ValueSet 中，为 a 和 b 分别赋值 1 和 2。而在 Observer 中，只是打印出 a 和 b 的值。可以想象，由于 Observer 打印 a 和 b 的时间与 ValueSet 设置 a 和 b 的时间可能有多种组合方式，因此 Obsever 可能打印出 (0, 0)，或者 (1, 2)，甚至是 (1, 0) 这样的结果。不过无论 Observer 打印了什么，在线程结束后再打印 a 和 b 的值，总会得到（1, 2）这样的结果。如图 6-4 所示，展示了这样的多种可能性。

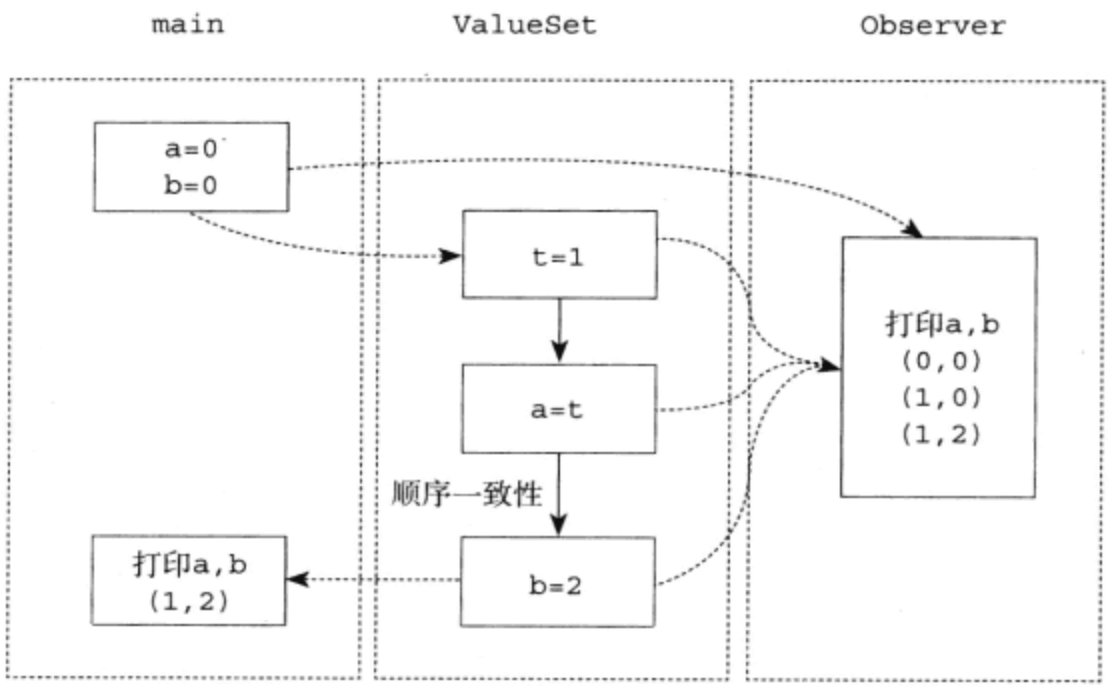


图 6-4　代码清单 6-21 所示例子的线程示意图

虽然 Observer 可能打印出 a、b 的 3 种组合，但这里如果 Observer 打印出 (0, 2) 这样的值是否合理呢？按照通常的程序是顺序执行的理解，(0,.2) 应该不是合理的输出。这从图 6-4 中也可以直观地看到，a 的赋值语句 a = t 总是先于 b 的赋值语句 b = 2 执行的，这是一个合乎情理的假设，但对于本例却并不重要。Observer 的编写者只是试图一窥线程 ValueSet 的执行状况，不过这种窥看相比于结果——线程结束后 a 和 b 的值总是 (1, 2) 而言，并不是必须的。也就是说，在本例的假定下，a、b 的赋值语句在 ValueSet 中谁先执行谁后执行并不会对程序的执行产生影响，因此说执行顺序是不重要的。

这一点假设虽然看似并不起眼，但对于编译器（甚至是处理器，下面我们会解释）来说非常重要。通常情况下，如果编译器认定 a、b 的赋值语句的执行先后顺序对输出结果有任何的影响的话，则可以依情况将指令重排序（reorder）以提高性能。而如果 a、b 赋值语句的执行顺序必须是 a 先 b 后，则编译器则不会执行这样的优化。如果我们假定，所有的原子类型的执行顺序都无关紧要，那么在多线程情况下就可能发生严重的错误。我们来看看代码清单

6-22 所示的例子。

代码清单 6-22

```
#include <thread>
#include <atomic>
#include <iostream>
using namespace std;

atomic<int> a;
atomic<int> b;

int Thread1(int) {
    int t = 1;
    a = t;
    b = 2;
}

int Thread2(int) {
    while(b != 2)
        ;  // 自旋等待
    cout << a << endl;  // 总是期待 a 的值为 1
}

int main() {
    thread t1(Thread1, 0);
    thread t2(Thread2, 0);

    t1.join();
    t2.join();
    return 0;
}

// 编译选项 :g++ -std=c++11 6-3-5.cpp -lpthread
```

在代码清单 6-22 中，Thread2 函数所在线程一开始总是在自旋等待，直到 b 的值被赋值为 2，它才会继续执行打印 a 的指令。如果这里，我们假设 Thread1 中 a 的赋值语句的执行被重排序到 b 的赋值语句之后的话，那么 Thread2 则可能打印出 a 的值为 0。这与程序员的看见的代码执行顺序完全背离，而一旦发生这样的情况，程序员也很难想象居然这是编译器（或者处理器）改变了代码的执行顺序而导致错误。因此为了避免这样的错误，在多线程情况下，非常有必要保证如同代码清单 6-21 及代码清单 6-22 中原子变量 a 的赋值语句先于原子变量 b 的赋值语句发生。

实际上默认情况下，在 C++11 中的原子类型的变量在线程中总是保持着顺序执行的特性（非原子类型则没有必要，因为不需要在线程间进行同步）。我们称这样的特性为“顺序一致”的，即代码在线程中运行的顺序与程序员看到的代码顺序一致，a 的赋值语句永远发

生于 b 的赋值语句之前。这样的“顺序一致”能够最大限度地保证程序的正确性。如同我们在代码清单 6-22 中看到的一样，a 的赋值语句先于 b 的赋值语句发生，这样的“先于发生”（happens-before）关系必须得到遵守，否则可能导致严重的错误。不过偏偏在代码清单 6-21 中我们又看到了相反的例子，ValueSet 中的 a、b 赋值语句的执行顺序并不重要。如果我们能够允许编译器（处理器）在单个线程中打乱指令的运行顺序，即不遵守先于发生的关系的话，则有可能进一步并行程序的性能。

那么有没有办法让一些代码遵守先于发生的关系，而另外一部分的代码不遵守呢？在 C++11 中，这是完全可能呢。不过语言的设计者的考量远远多过于这一点。更为确切地，他们对各种平台、处理器、编程方式都进行了考量，总结出了不同的“内存模型”。事实上，顺序一致只是属于 C++11 中多种内存模型中的一种。而在 C++11 中，并不是只支持顺序一致单个内存模型的原子变量，因为顺序一致往往意味着最低效的同步方式。要使用 C++11 中更为高效的原子类型变量的同步方式，我们先要了解一些处理器和编译器相关的知识。

通常情况下，内存模型通常是一个硬件上的概念，表示的是机器指令（或者读者将其视为汇编语言指令也可以）是以什么样的顺序被处理器执行的。现代的处理器并不是逐条处理机器指令的，我们可以看看下面这个段伪汇编码：

```
1: Loadi     reg3, 1;     # 将立即数 1 放入寄存器 reg3
2: Move     reg4, reg3;   # 将 reg3 的数据放入 reg4
3: Store     reg4, a;     # 将寄存器 reg4 中的数据存入内存地址 a
4: Loadi     reg5, 2;     # 将立即数 2 放入寄存器 reg5
5: Store     reg5, b;     # 将寄存器 reg5 中的数据存入内存地址 b
```

这里我们演示了“t = 1; a = t; b = 2;”这段 C++ 语言代码的伪汇编表示。按照通常的理解，指令总是按照 1->2->3->4->5 这样顺序执行，如果处理器的执行顺序是这样的话，我们通常称这样的内存模型为强顺序的（strong ordered）。可以看到，在这种执行方式下，指令 3 的执行（a 的赋值）总是先于指令 5（b 的赋值）发生。

不过这里我们看到，指令 1、2、3 和指令 4、5 运行顺序上毫无影响（使用了不同的寄存器，以及不同的内存地址），一些处理器就有可能将指令执行的顺序打乱，比如按照 1->4->2->5->3 这样顺序（通常这样的执行顺序都是超标量的流水线，即一个时钟周期里发射多条指令而产生的）。如果指令是按照这个顺序被处理器执行的话，我们通常称之为弱顺序的（weak ordered）。而在这种情况下，指令 5（b 的赋值）的执行可能就被提前到指令 3（a 的赋值）完成之前完成。

注意 事实上，一些弱内存模型的构架比如 PowerPC，其写回操作是不能够被乱序的，这里只是一个帮助读者理解的示例，并非事实。

那么在多线程情况下，强顺序和弱顺序又意味着什么呢？我们知道，多线程的程序总是共享代码的，那么强顺序意味着：对于多个线程而言，其看到的指令执行顺序是一致的。具

体地，对于共享内存的处理器而言，需要看到内存中的数据被改变的顺序与机器指令中的一致。反之，如果线程间看到的内存数据被改变的顺序与机器指令中声明的不一致的话，则是弱顺序的。比如在我们的伪汇编中，假设运行的平台遵从的是一个弱顺序的内存模型的话，那么可能线程 A 所在的处理器看到指令执行顺序是先 3 后 5，而线程 B 以为指令执行的顺序依然是先 5 后 3，那么反馈到代码清单 6-22 的源代码中，我们就有可能看 Thread2 打印出的 a 的值是 0 了。

在现实中，x86 以及 SPARC（TSO 模式）都被看作是采用强顺序内存模型的平台。对于任何一个线程而言，其看到原子操作（这里都是指数据的读写）都是顺序的。而对于是采用弱顺序内存模型的平台，比如 Alpha、PowerPC、Itanlium、ArmV7 这样的平台而言，如果要保证指令执行的顺序，通常需要由在汇编指令中加入一条所谓的内存栅栏（memory barrier）指令。比如在 PowerPC 上，就有一条名为 sync 的内存栅栏指令。该指令迫使已经进入流水线中的指令都完成后处理器才执行 sync 以后指令（排空流水线）。这样一来，sync 之前运行的指令总是先于 sync 之后的指令完成的。比如我们可以这样来保证我们伪汇编中的指令 3 的执行先于指令 5：

```
1: Loadi    reg3, 1;     # 将立即数 1 放入寄存器 reg3
2. Move    reg4, reg3;   # 将 reg3 的数据放入 reg4
3: Store    reg4, a;     # 将寄存器 reg4 中的数据存入内存地址 a
4: Sync                  # 内存栅栏
5: Loadi    reg5, 2;     # 将立即数 2 放入寄存器 reg5
6: Store    reg5, b;     # 将寄存器 reg5 中的数据存入内存地址 b
```

sync 指令对高度流水化的 PowerPC 处理器的性能影响很大，因此，如果可以不顺序提交语句的运行结果的话，则可以保证弱顺序内存模型的处理器保持较高的流水线吞吐率（throughput）和运行时性能。

注意 为什么会有弱顺序的内存模型？

简单地说，弱顺序的内存模型可以使得处理器进一步发掘指令中的并行性，使得指令执行的性能更高。

注意 为什么我们只关心读写操作的执行顺序问题？

这是由处理器的设计决定的，通常情况下，处理器总是从内存中读出数据进行运算，再将运行结果又返回内存，因此内存中的数据是一个“准绳”，相对的，寄存器中的内容则是“临时量”。所以在多核心处理器上，核心往往都有全套的寄存器来分别存储临时量，而数据交流总是以内存中的数据为准。这么一来，一些寄存器中的运算（比如伪汇编中的指令 2）就不会被多处理器关注，处理器只关心读写等原子操作指令的顺序。

以上都是硬件上一些可能的内存模型的描述。而 C++11 中定义的内存模型和顺序一致性

跟硬件的内存模型的强顺序、弱顺序之间有着什么样的联系呢？事实上，在高级语言和机器指令间还有一层隔离，这层隔离是由编译器来完成的。如我们之前描述的，编译器出于代码优化的考虑，会将指令前后移动，已获得最佳的机器指令的排列及产生最佳的运行时性能。那么对于 C++11 中的内存模型而言，要保证代码的顺序一致性，就必须同时做到以下几点：

- 编译器保证原子操作的指令间顺序不变，即保证产生的读写原子类型的变量的机器指令与代码编写者看到的是一致的。
- 处理器对原子操作的汇编指令的执行顺序不变。这对于 x86 这样的强顺序的体系结构而言，并没有任何的问题；而对于 PowerPC 这样的弱顺序的体系结构而言，则要求编译器在每次原子操作后加入内存栅栏。

如前文所述，在 C++11 中，原子类型的成员函数（原子操作）总是保证了顺序一致性。这对于 x86 这样的平台来说，禁止了编译器对原子类型变量间的重排序优化；而对于 PowerPC 这样的平台来说，则不仅禁止了编译器的优化，还插入了大量的内存栅栏。这对于意图是提高性能的多线程程序而言，无疑是一种性能伤害。具体而言，对于代码清单 6-21 中 ValueSet 这样的不需要遵守 a、b 赋值语句“先于发生”关系的程序而言，由于 atomic 默认的顺序一致性则会在对 a、b 的赋值语句间加入内存栅栏，并阻止编译器优化，这无疑会增加并行开销（内存栅栏尤其如此）。那么解除这样的性能约束也势在必行。

在 C++11 中，设计者给出的解决方式是让程序员为原子操作指定所谓的内存顺序：memory_order。比如在代码清单 6-21 中，就可以采用一种松散的内存模型（relaxed memory model）来放松对原子操作的执行顺序的要求。我们来看看代码清单 6-23 对代码清单 6-21 的所作的改进。

代码清单 6-23

```
#include <thread>
#include <atomic>
#include <iostream>
using namespace std;

atomic<int> a {0};
atomic<int> b {0};

int ValueSet(int) {
    int t = 1;
    a.store(t, memory_order_relaxed);
    b.store(2, memory_order_relaxed);
}

int Observer(int) {
    cout << "(" << a << ", " << b << ")" << endl;    // 可能有多种输出
}
```

```
int main() {
    thread t1(ValueSet, 0);
    thread t2(Observer, 0);

    t1.join();
    t2.join();
    cout << "Got (" << a << ", " << b << ")" << endl;    // Got (1, 2)
    return 0;
}

// 编译选项 :g++ -std=c++11 6-3-6.cpp -lpthread
```

在代码清单 6-23 中，我们对 ValueSet 函数进行了改造。之前的对 a、b 进行赋值的语句我们改用了 atomic 类模板的 store 成员。store 能够接受两个参数，一个是需要写入的值，一个是名为 memory_order 的枚举值。这里我们使用的值是 memory_order_relaxed，表示使用松散的内存模型，该指令可以任由编译器重排序或者由处理器乱序执行。这样一来，a、b 赋值语句的“先于发生”顺序得到了解除，我们也就可能得到最佳的运行性能。当然，相应的结果是，对于 Observer 来说，打印出 (0, 2) 这样的结果也就是合理的了。

如我们在上节最后提到的，大多数 atomic 原子操作都可以使用 memory_order 作为一个参数，在 C++11 中，标准一共定义了 7 种 memory_order 的枚举值，如表 6-3 所示。

表 6-3　C++11 中的 memory_order 枚举值

枚举值	定义规则
memory_order_relaxed	不对执行顺序做任何保证
memory_order_acquire	本线程中，所有后续的读操作必须在本条原子操作完成后执行
memory_order_release	本线程中，所有之前的写操作完成后才能执行本条原子操作
memory_order_acq_rel	同时包含 memory_order_acquire 和 memory_order_release 标记
memory_order_consume	本线程中，所有后续的有关本原子类型的操作，必须在本条原子操作完成之后执行
memory_order_seq_cst	全部存取都按顺序执行

memory_order_seq_cst 表示该原子操作必须是顺序一致的，这是 C++11 中所有 atomic 原子操作的默认值，不带 memory_order 参数的原子操作就是使用该值。而 memorey_order_relaxed 则表示该原子操作是松散的，可以被任意重排序的。其他几种我们会在后面解释。值得注意的是，并非每种 memory_order 都可以被 atomic 的成员使用。通常情况下，我们可以把 atomic 成员函数可使用的 memory_order 值分为以下 3 组：

- 原子存储操作（store）可以使用 memorey_order_relaxed、memory_order_release、memory_order_seq_cst。
- 原子读取操作（load）可以使用 memorey_order_relaxed、memory_order_consume、memory_order_acquire、memory_order_seq_cst。

- RMW 操作（read-modify-write），即一些需要同时读写的操作，比如之前提过的 atomic_flag 类型的 test_and_set() 操作。又比如 atomic 类模板的 atomic_compare_exchange() 操作等都是需要同时读写的。RMW 操作可以使用 memorey_order_relaxed、memory_order_consume、memory_order_acquire、memory_order_release、memory_order_acq_rel、memory_order_seq_cst。

一些形如“operator =”、“operator +=”的函数，事实上都是 memory_order_seq_cst 作为 memory_order 参数的原子操作的简单封装。也即是说，之前小节中的代码都是采用顺序一致性的内存模型。如果读者需要的正是顺序一致性的内存模型的话，那么这些操作符都是可以直接使用的。而如果读者是要指定内存顺序的话，则应该采用形如 load、atomic_fetch_add 这样的版本。

如之前提到的，memory_order_seq_cst 这种 memory_order 对于 atomic 类型数据的内存顺序要求过高，容易阻碍系统发挥线程应有的性能。而 memorey_order_relaxed 对内存顺序毫无要求，这在代码清单 6-21 中满足了我们解除“先于发生”顺序的需求。但在另外一些情况下，则还是可能无法满足真正的需求。我们可以看看由代码清单 6-22 改造而来的代码清单 6-24 的例子。

代码清单 6-24

```
#include <thread>
#include <atomic>
#include <iostream>
using namespace std;

atomic<int> a;
atomic<int> b;

int Thread1(int) {
    int t = 1;
    a.store(t, memory_order_relaxed);
    b.store(2, memory_order_relaxed);
}

int Thread2(int) {
    while(b.load(memory_order_relaxed) != 2);  // 自旋等待
    cout << a.load(memory_order_relaxed) << endl;
}

int main() {
    thread t1(Thread1, 0);
    thread t2(Thread2, 0);

    t1.join();
    t2.join();
    return 0;
```

```
}

// 编译选项 :g++ -std=c++11 6-3-7.cpp -lpthread
```

代码清单 6-24 与代码清单 6-22 的例子基本一致，只不过这里我们并不希望完全禁用关于原子类型的优化，而采用了 memory_order_relaxed 作为 memory_order 参数。在一些弱内存模型的机器上，这两条 a、b 赋值语句将有可能任意一条被先执行。那么对于 Thread2 函数而言，它先是自旋等待 b 的值被赋为 2，随后将 a 的值输出。按照松散的内存顺序，我们输出的 a 的值则有可能为 0，也有可能为 1。这显然是不符合代码作者的期望的。

那么排除顺序一致和松散两种方式，我们能不能保证程序“既快又对”地运行呢？如果读者仔细地分析的话，我们所需要的只是 a.store 先于 b.store 发生，b.load 先于 a.load 发生的顺序。这要这两个“先于发生”关系得到了遵守，对于整个程序而言来说，就不会发生线程间的错误。建立这种“先于发生”关系，即原子操作间的顺序则需要利用其他的 memory_order 枚举值。我们可以看看代码清单 6-25 中修改的代码。

代码清单 6-25

```
#include <thread>
#include <atomic>
#include <iostream>
using namespace std;

atomic<int> a;
atomic<int> b;

int Thread1(int) {
    int t = 1;
    a.store(t, memory_order_relaxed);
    b.store(2, memory_order_release); // 本原子操作前所有的写原子操作必须完成
}

int Thread2(int) {
     while(b.load(memory_order_acquire) != 2);   // 本原子操作必须完成才能执行之后所有的读原子操作
    cout << a.load(memory_order_relaxed) << endl;    // 1
}

int main() {
    thread t1(Thread1, 0);
    thread t2(Thread2, 0);

    t1.join();
    t2.join();
    return 0;
}
```

```
// 编译选项 :g++ -std=c++11 6-3-8.cpp -lpthread
```

这里代码清单 6-25 对代码清单 6-24 做了两处改动，一是 b.store 采用了 memory_order_release 内存顺序，这保证了本原子操作前所有的写原子操作必须完成，也即 a.store 操作必须发生于 b.store 之前。二是 b.load 采用了 memory_order_acquire 作为内存顺序，这保证了本原子操作必须完成才能执行之后所有的读原子操作。即 b.load 必须发生在 a.load 操作之前。这样一来，通过确立“先于发生”关系的，我们就完全保证了代码运行的正确性，即当 b 的值为 2 的时候，a 的值也确定地为 1。而打印语句也不会在自旋等待之前打印 a 的值。Thread1 和 Thread2 的执行顺如图 6-5 所示。

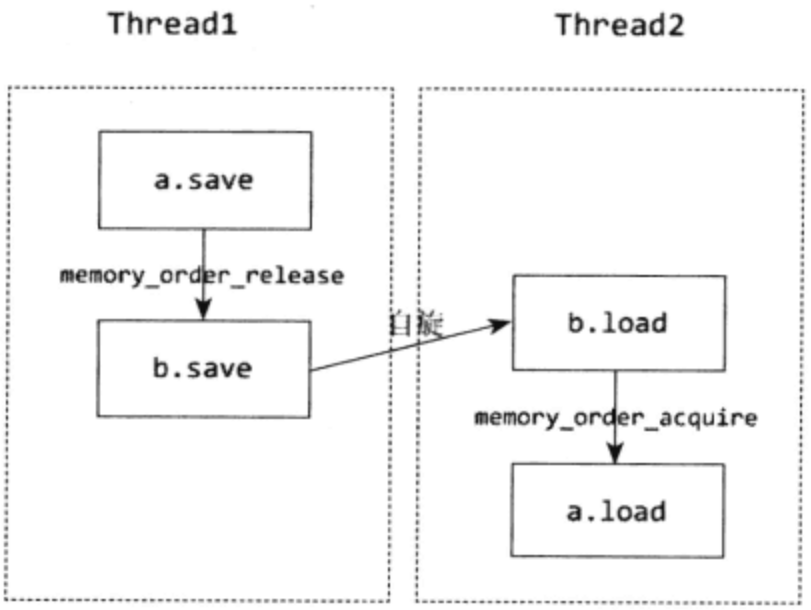


图 6-5 Thread1 和 Thread2 的顺序

由于 memory_order_release 和 memory_order_acquire 常常结合使用，我们也称这种内存顺序为 release-acquire 内存顺序。

通常情况下，“先于发生”关系总是传递的，比如原子操作 A 发生于原子操作 B 之前，而原子操作 B 又发生于原子操作 C 之前的话，则 A 一定发生于 C 之前。有了这样的顺序，就可以指导编译器在重排序指令的时候在不破坏依赖规则（相当于多给了一些依赖关系）的情况下，仅在适当的位置插入内存栅栏，以保证执行指令时数据执行正确的同时获得最佳的运行性能。

在表 6-3 中，我们还看到了 memory_order_consume 这个 memory_order 的枚举值。该枚举值与 memory_order_acquire 相比，进一步放松了一些依赖关系。我们可以看看代码清单 6-26 所示的例子[⊖]。

⊖ 本例来自于 http://en.cppreference.com/w/cpp/atomic/memory_order。

代码清单 6-26

```
#include <thread>
#include <atomic>
#include <cassert>
#include <string>
using namespace std;

atomic<string*> ptr;
atomic<int> data;

void Producer() {
    string* p  = new string("Hello");
    data.store(42, memory_order_relaxed);
    ptr.store(p, memory_order_release);
}

void Consumer() {
    string* p2;
    while (!(p2 = ptr.load(memory_order_consume)))
        ;
    assert(*p2 == "Hello");      // 总是相等
    assert(data.load(memory_order_relaxed) == 42);  // 可能断言失败
}

int main() {
    thread t1(Producer);
    thread t2(Consumer);

    t1.join();
    t2.join();
}

// 编译选项 :g++ -std=c++11 6-3-9.cpp -lpthread
```

在代码清单 6-26 中，我们定义了两个线程 t1 和 t2，分别运行 Producer 和 Consumer 函数。在 Producer 函数中，使用了 memory_order_release 来为原子类型 atomic<string*> 变量 ptr 存储一个值；而在 Consumer 函数中，通过 memory_order_consume 的内存顺序来完成变量 ptr 的读取。这里我们可以看到，这样的内存顺序保证了 ptr.load(memory_order_consume) 必须发生在 *ptr 这样的解引用操作（实际上涉及的是读指针 ptr.load 的操作）之前。不过与 memory_order_acquire 不同的是，该操作并不保证发生在 data.load(memory_order_relaxed) 之前，因为 data 和 ptr 是不同的原子类型数据，而 memory_order_comsume 只保证原子操作发生在与 ptr 有关的原子操作之前。所以实际上相比于 memory_order_acquire，“先于发生”的关系又被弱化了。

形如其名，memory_order_release 和 memory_order_consume 的配合会建立关于原子类型的“生产者 - 消费者”的同步顺序。同样的，我们可以称之为 release-consume 内存顺序。

顺序一致、松散、release-acquire 和 release-consume 通常是最为典型的 4 种内存顺序。其他的如 memory_order_acq_rel，则是常用于实现一种叫做 CAS（compare and swap）的基本同步元语，对应到 atomic 的原子操作 compare_exchange_strong 成员函数上。我们也称之为 acquire-release 内存顺序。

事实上，由于并行编程在 C++11 中是非常新的一个话题，因此 C++11 中关于原子操作的设计还涉及大量的细节和众多特性。不过在本书编写时，还没有编译器正式支持所有并行的特性，为了避免理解上的偏差，因此除去语言中关于并行的比较核心的部分，本书也不再进一步地进行讲解了[⊖]。

而回到内存模型上来。虽然在 C++11 中，我们看到了大量的内存顺序相关的设计。不过这样的设计主要还是为了从各种繁杂不同的平台上抽象出独立于硬件平台的并行操作。如果读者不太愿意了解内存模型等相关概念，那么简单地使用 C++11 原子操作的顺序一致性就可以进行并行程序的编写了。而如果读者想让自己的程序在多线程情况下获得更好的性能的话，尤其当使用的是一些弱内存顺序的平台，比如 PowerPC 的话，建立原子操作间内存顺序则很有必要，因为这可会带来极大的性能提升（事实上，这也是弱一致性内存模型平台的优势）。

但对于并行编程来说，可能最根本的（这是本书没有涉及的话题）还是思考如何将大量计算的问题，按需分解成多个独立的、能够同时运行的部分，并找出真正需要在线程间共享的数据，实现为 C++11 的原子类型。虽然有了原子类型的良好设计，实现这些都可以非常的便捷，但并不是所有的问题或者计算都适合用并行计算来解决，对于不适用的问题，强行用并行计算来解决会收效甚微，甚至起到相反效果。因此在决定使用并行计算解决问题之前，程序员必须要有清晰的设计规划。而在实现了代码并行后，进一步使用一些性能调试工具来提高并行程序的性能也是非常必要的。

6.4 线程局部存储

☞ **类别：所有人**

线程局部存储（TLS, thread local storage）是一个已有的概念。简单地说，所谓线程局部存储变量，就是拥有线程生命期及线程可见性的变量。

线程局部存储实际上是由单线程程序中的全局 / 静态变量被应用到多线程程序中被线程共享而来。我们可以简单地回顾一下所谓的线程模型。通常情况下，线程会拥有自己的栈空

⊖ 我们从 svn 上得到的 gcc 版本应该对应的是 gcc-4.8，gcc-4.8 可能会支持所有并行的语义。不过本书编写时，gcc-4.8 还没有发布。

间，但是堆空间、静态数据区（如果从可执行文件的角度来看，静态数据区对应的是可执行文件的 data、bss 段的数据，而从 C/C++ 语言层面而言，则对应的是全局 / 静态变量）则是共享的。这样一来，全局、静态变量在这种多线程模型下就总是在线程间共享的。

全局、静态变量的共享虽然会带来一些好处，尤其对一些资源性的变量（比如文件句柄）来说也是应该的，不过并不是所有的全局、静态变量都适合在多线程的情况下共享。我们可以看看代码清单 6-27 所示的例子。

代码清单 6-27

```
#include <pthread.h>
#include <iostream>
using namespace std;

int errorCode = 0;

void* MaySetErr(void * input) {
    if (*(int*)input == 1)
        errorCode = 1;
    else if (*(int*)input == 2)
        errorCode = -1;
    else
        errorCode = 0;
}
int main() {
    int input_a = 1;
    int input_b = 2;

    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, &MaySetErr, &input_a);
    pthread_create(&thread2, NULL, &MaySetErr, &input_b);

    pthread_join(thread2, NULL);
    pthread_join(thread1, NULL);
    cout << errorCode << endl;
}

// 编译选项 :g++ 6-4-1.cpp -lpthread
```

在代码清单 6-27 中，函数 MaySetErr 函数可能会根据输入值 input 设置全局的错误码 errorCode。当用两个线程运行该函数的时候，最终获得的 errorCode 的值将是不确定的，或者说，将由系统如何调度两个线程而决定。实际上，本例中的 errorCode 即是 POSIX 标准中的错误码全局变量 errno 在多线程情况下遭遇的问题的一个简化。一旦 errno 在线程间共享，则一些程序中运行的错误将会被隐藏不报。而解决的办法就是为每个线程指派一个全局的 errno，即 TLS 化的 errno。

各个编译器公司都有自己的 TLS 标准。我们在 g++/clang++/xlc++ 中可以看到如下的语法：

```
__thread int errCode;
```

即在全局或者静态变量的声明中加上关键字 __thread，即可将变量声明为 TLS 变量。每个线程将拥有独立的 errCode 的拷贝，一个线程中对 errCode 的读写并不会影响另外一个线程中的 errCode 的数据。

C++11 对 TLS 标准做出了一些统一的规定。与 __thread 修饰符类似，声明一个 TLS 变量的语法很简单，即通过 thread_local 修饰符声明变量即可。

```
int thread_local errCode;
```

一旦声明一个变量为 thread_local，其值将在线程开始时被初始化，而在线程结束时，该值也将不再有效。对于 thread_local 变量地址取值（&），也只可以获得当前线程中的 TLS 变量的地址值。

虽然 TLS 变量的声明很简单，使用也很直观，不过实际上 TLS 的实现需要涉及编译器、链接器、加载器甚至是操作系统的相互配合。在 TLS 中一个常被讨论的问题就是 TLS 变量的静态 / 动态分配的问题，即 TLS 变量的内存究竟是在程序一开始就被分配还是在线程开始运行时被分配。通常情况下，前者比后者更易于实现。C++11 标准允许平台 / 编译器自行选择采用静态分配或动态分配，或者两者都支持。还有一点值得注意的是，C++11 对 TLS 只是做了语法上的统一，而对其实现并没有做任何性能上的规定。这可能导致 thread_local 声明的变量在不同平台或者不同的 TLS 实现上出现不同的性能（通常 TLS 变量的读写性能不会高于普通的全局 / 静态变量）。如果读者想得到最佳的平台上的 TLS 变量的运行性能的话，最好还是阅读代码运行平台的相关文档。

6.5 快速退出：quick_exit 与 at_quick_exit

☞ **类别：所有人**

在 C++ 程序中，我们常常会看到一些有关“终止”的函数，如 terminate、abort、exit 等。这些函数容易让人产生疑惑，因为对于普通的程序来说，它们都只是终止程序的运行而已。不过实际上它们还是有很大的区别的。因为其对应的是“正常退出”和“异常退出”两种情况。

首先我们可以看看 terminate 函数，terminate 函数实际上是 C++ 语言中异常处理的一部分（包含在 <exception> 头文件里）。一般而言，没有被捕捉的异常就会导致 terminate 函数的调用。此外我们在第 2 章中提到过的 noexcept 关键字声明的函数，如果抛出了异常，也会调用 terminate 函数。其他还有很多的情况。但直观地讲，只要 C++ 程序中出现了非程序员预期的行为，都有可能导致 terminate 的调用。而 terminate 函数在默认情况下，是去调用 abort

函数的。不过用户可以通过 set_terminate 函数来改变默认的行为。因此，可以认为在 C++ 程序的层面，termiante 就是“终止”。

相对于 termiante，源自于 C 中（头文件 <cstdlib>）的 abort 则更加低层。abort 函数不会调用任何的析构函数（读者也许想到了，默认的 terminate 也是如此），默认情况下，它会向合乎 POSIX 标准的系统抛出一个信号（signal）：SIGABRT。如果程序员为信号设定一个信号处理程序的话（signal handler），那么操作系统将默认地释放进程所有的资源，从而终止程序。可以说，abort 是系统在毫无办法下的下下策——终止进程。有时候这会带来一些问题。典型的，倘若被终止的应用程序进程与其他应用程序软件层有一些交互（比如一些硬件驱动程序，一些通过网络通信的程序等，假设这些程序设计得并不那么健壮），那么本进程的意外终止，都可能导致这些交互进程处于一些“中间状态”，进而出现一些问题。

相比而言，exit 这样的属于“正常退出”范畴的程序终止，则不太可能有以上的问题。exit 函数会正常调用自动变量的析构函数，并且还会调用 atexit 注册的函数。这跟 main 函数结束时的清理工作是一样的。我们可以看看代码清单 6-28 所示的例子。

代码清单 6-28

```
#include <cstdlib>
#include <iostream>
using namespace std;

void openDevice() { cout << "device is opened." << endl; }

void resetDeviceStat() { cout << "device stat is reset." << endl; }

void closeDevice() { cout << "device is closed." << endl; }

int main() {
    atexit(closeDevice);
    atexit(resetDeviceStat);
    openDevice();
    exit(0);
}
```

编译选项：g++ 6-5-1.cpp

在代码清单 6-28 中，我们使用 atexit 注册了两个函数：resetDeviceStat 和 closeDevice。编译运行该例子后，程序的输出如下：

```
device is opened.
device stat is reset.
device is closed.
```

可以看到，在程序退出时（调用 ANSI C 定义的 exit 函数的时候），所有注册的函数都被

调用，值得注意的是，注册的函数被调用的次序与其注册顺序相反，这多少跟析构函数的执行与其声明的顺序相反是一致的。exit 和 atexit 函数同样来自于 C，通过两者的配合，我们可以灵活地处理一些进程级的清理工作，这对一些静态、全局变量来说，是非常有用的。

不过有的时候，main 或者使用 exit 函数调用结束程序的方式也不那么令人满意。有的时候，代码中会有很多的类，这些类在堆空间上分配了大量的零散的内存（直接从堆里分配，并没有优化的策略），而 main 或者 exit 函数调用会导致类的析构函数依次将这些零散的内存还给操作系统。这是一件费时的工作，而实际上，这些堆内存将在进程结束时由操作系统统一回收（事实上这相当快，操作系统除了释放一些进程相关的数据结构外，只是将一些物理内存标记为未使用就可以了）。如果这些堆内存对其他程序不产生任何影响，那么在程序结束时释放堆内存的析构过程往往是毫无意义的。因此，在这种情况下，我们常常需要能够更快地退出程序。

另外，在多线程情况下，我们要使用 exit 函数来退出程序的话，通常需要向线程发出一个信号，并等待线程结束后再执行析构函数、atexit 注册的函数等。这从语法上讲非常正确，不过这样的退出方式有的时候并不能够像预期那样工作，比如说线程中的程序在等待 I/O 运行结束等。在一些更为复杂的情况下，可能还会遭遇到一些因为信号顺序而导致的死锁状况。一旦出现了这样的问题，程序往往就会被“卡死”而无法退出。

为此，在 C++11 中，标准引入了 quick_exit 函数，该函数并不执行析构函数而只是使程序终止。与 abort 不同的是，abort 的结果通常是异常退出（可能系统还会进行 coredump 等以辅助程序员进行问题分析），而 quick_exit 与 exit 同属于正常退出。此外，使用 at_quick_exit 注册的函数也可以在 quick_exit 的时候被调用。这样一来，我们同样可以像 exit 一样做一些清理的工作（这与很多平台上使用 _exit 函数直接正常退出还是有不同的）。在 C++11 标准中，at_quick_exit 和 at_exit 一样，标准要求编译器至少支持 32 个注册函数的调用。代码清单 6-29 所示是一个可能能够运行的例子。

代码清单 6-29

```
#include <cstdlib>
#include <iostream>
using namespace std;

struct A { ~A() { cout << "Destruct A. " << endl; } };

void closeDevice() { cout << "device is closed." << endl; }

int main() {
    A a;
    at_quick_exit(closeDevice);
    quick_exit(0);
}
```

这里我们定义了一个类型A的变量a，以及注册了一个quick_exit调用的函数closeDevice。如果示例正确的话，变量a的析构函数将不会被调用。

6.6 本章小结

在本章中，我们首先看到了C++11中与性能极其相关的新特性：常量表达式constexpr。常量表达式通常可以用于修饰函数、变量以及构造函数等，以使得声明constexpr关键字的函数和变量可以被用于编译时的值计算。这样一来，一些本是运行时常量的运算却可以被合理地放到编译时，而一些语法上的限制也被常量表达式解放了出来。而由constexpr演化出的constexpr元编程则成了C++中继模板元编程之后又一个可以进行编译时值计算的手段。而其超越模板元编程的各种优势，使得其应用前景被广泛看好。

变长模板是C++引入的新的“变长”参数的工具，不过远胜于变长宏和变长函数。变长模板通过模板偏特化以及一些递归引用的定义，可以在不丢失类型信息的情况下实现变长参数的传递。从某种意义上讲，变长模板把泛型编程又推向了一个新的高度，也使得变长模板在库编写的领域有着很好的应用。

而原子操作则彻底宣告C++来到了并行编程和多线程的时代。相比于偏于底层的pthread库，C++通过定义原子类型的方式，轻松地化解了互斥访问同步变量的难题。不过C++也延续了其易于学习难于精通的特性。虽然原子类型使用很简单，但其成员变量（原子操作）却可以有各种不同的内存顺序。C++11从各种不同的平台上抽象出了一个软件的内存模型，并以内存顺序进行描述，以使得想进一步挖掘并行系统性能的程序员有足够简单的手段来完成以往只能通过内联汇编来完成的工作。这样的高度总结和设计，也堪称C++11中的一大亮点。

此外，为了适应并行编程，C++11还将广泛存在的线程局部存储进行了语法上的统一。并且标准也为TLS留下了足够的余地，以适应于各种平台的TLS的实现。quick_exit则是一项多线程情况下的新发明，可以用于解除因为退出造成的死锁等不良状态。不过读者也可以尝试着使用它来免除大量的不必要的析构函数调用。

总的看来，C++11除了突破自身语法，除了在泛型编程的技巧上更上一层楼外，也延续了C和以前C++版本对硬件操控的强能力，而且将这种能力带入了并行和多线程的时代。虽然这可能带来一些学习的代价，不过这些能力常常是其他语言难以具备的。因此，真正了解这些新特性，可以让使用者可以更轻松地完成各种复杂的工作。对于一些程序员来讲（比如库开发人员，系统级的程序员），这是一种简化，而不是复杂化。

第⑦章

为改变思考方式而改变

如我们提到过的，C++ 是一门成熟的语言，语言的核心部分的改变通常都遵从着一贯的设计思想。不过这并不意味着 C++ 会墨守成规，在 C++11 中，我们还是会看到一些新元素。这些新鲜出炉的元素可能会带来一些习惯上的改变，不过权衡之下，可能这样的改变是值得的。比如 lambda 就是一个典型的例子。

7.1 指针空值——nullptr

☞ 类别：所有人

7.1.1 指针空值：从 0 到 NULL，再到 nullptr

在良好的 C++ 编程习惯中，声明一个变量的同时，总是需要记得在合适的代码位置将其初始化。对于指针类型的变量，这一点尤其应当注意。未初始化的悬挂指针通常会是一些难于调试的用户程序的错误根源。

典型的初始化指针是将其指向一个“空”的位置，比如 0。由于大多数计算机系统不允许用户程序写地址为 0 的内存空间，倘若程序无意中对该指针所指地址赋值，通常在运行时就会导致程序退出。虽然程序退出并非什么好事，但这样一来错误也容易被程序员找到。因此在大多数的代码中，我们常常能看见指针初始化的语法如下：

```
int * my_ptr = 0;
```

或者是使用 NULL：

```
int * my_ptr = NULL;
```

一般情况下，NULL 是一个宏定义。在传统的 C 头文件（stddef.h) 里我们可以找到如下代码：

```
#undef NULL
#if defined(__cplusplus)
#define NULL 0
#else
#define NULL ((void *)0)
#endif
```

可以看到，NULL 可能被定义为字面常量 0，或者是定义为无类型指针（void *）常量。不过无论采用什么样的定义，我们在使用空值的指针时，都不可避免地会遇到一些麻烦。让我们先看一个关于函数重载的例子。这个例子我们引用自 C++11 标准关于 nullptr 的提案，并进行了少许修改，具体如代码清单 7-1 所示。

代码清单 7-1

```
#include <stdio.h>

void f(char* c) {
    printf("invoke f(char*)\n");
}

void f(int i) {
    printf("invoke f(int)\n");
}

int main() {
    f(0);
    f(NULL); // 注意：如用 gcc 编译，NULL 转化为内部标识 __null，该语句会编译失败
    f((char*)0);
}

// 编译选项 :xlC -+ 7-1-1.cpp
```

在代码清单 7-1 所示的例子当中，用户重载了 f 函数，并且试图使用 f(NULL) 来调用指针的版本。不过很可惜，当使用 XLC 编译器编译以上语句并运行时，会得到以下结果：

```
invoke f(int)
invoke f(int)
invoke f(char*)
```

在这里，XLC 编译器采用了 stddef.h 头文件中 NULL 的定义，即将 NULL 定义为 0。因此使用 NULL 做参数调用和使用字面量 0 做参数调用版本的结果完全相同，都是调用到了 f(int) 这个版本。这实际与程序员编写代码的意图相悖。

引起该问题的元凶是字面常量 0 的二义性，在 C++98 标准中，字面常量 0 的类型既可以是一个整型，也可以是一个无类型指针（void* ）。如果程序员想在代码清单 7-1 中调用 f(char*) 版本的话，则必须像随后的代码一样，对字面常量 0 进行强制类型转换（(void*)0 ）并调用，否则编译器总是会优先把 0 看作是一个整型常量。

虽然这个问题可以通过修改代码来解决，但为了避免用户使用上的错误，有的编译器做了比较激进的改进。典型的如 g++ 编译器，它直接将 NULL 转换为编译器内部标识（ __null ），并在编译时期做了一些分析，一旦遇到二义性就停止编译并向用户报告错误。虽然这在一定程度上缓解了二义性带来的麻烦，但由于标准并没有认定 NULL 为一个编译时期的标识，所

以也会带来代码移植性的限制。

注意 关于 nullptr 和 void* 的翻译，void* 习惯被翻作无类型指针，我们这里把 nullptr 翻作指针空值。

在 C++11 新标准中，出于兼容性的考虑，字面常量 0 的二义性并没有被消除。但标准还是为二义性给出了新的答案，就是 nullptr。在 C++11 标准中，nullptr 是一个所谓“指针空值类型”的常量。指针空值类型被命名为 nullptr_t，事实上，我们可以在支持 nullptr 的头文件（cstddef）中找出如下定义：

```
typedef decltype(nullptr) nullptr_t;
```

可以看到，nullptr_t 的定义方式非常有趣，与传统的先定义类型，再通过类型声明值的做法完全相反（充分利用了 decltype 的功能）。我们发现，在现有编译器情况下，使用 nullptr_t 的时候必须 #include<cstddef>（#include 有些头文件也会间接 #include<cstddef>，比如 <iostream>），而 nullptr 则不用。这大概就是由于 nullptr 是关键字，而 nullptr_t 是通过推导而来的缘故。

而相比于 gcc 等编译器将 NULL 预处理为编译器内部标识 __null，nullptr 拥有更大的优势。简单而言，由于 nullptr 是有类型的，且仅可以被隐式转化为指针类型，那么对于代码 7-1 的例子，nullptr 做参数则可以成功调用 f(char*) 版本的函数，而不是像 gcc 对 NULL 的处理一样，仅仅给出一个出错提示，好让程序员去修改代码。

我们来看看代码清单 7-2 所示的例子。

代码清单 7-2

```
#include <iostream>
using namespace std;

void f(char *p) {
    cout << "invoke f(char*)" << endl;
}

void f(int) {
    cout << "invoke f(int)" << endl;
}

int main()
{
    f(nullptr);     // 调用 f(char*) 版本
    f(0);           // 调用 f(int) 版本
    return 0;
}

// 编译选项 :g++ 7-1-2.cpp -std=c++11
```

可以看到，在改为使用 nullptr 之后，用户能够准确表达自己的意图，也不会再出现在 XLC 编译器上调用了 f(int) 版本而在 gcc 上却在编译时期给出了错误提示的不兼容问题。因此，通常情况下，在书写 C++11 代码想使用 NULL 的时候，将 NULL 替换成为 nullptr 我们就能获得更加健壮的代码。

7.1.2 nullptr 和 nullptr_t

C++11 标准不仅定义了指针空值常量 nullptr，也定义了其指针空值类型 nullptr_t，也就表示了指针空值类型并非仅有 nullptr 一个实例。通常情况下，也可以通过 nullptr_t 来声明一个指针空值类型的变量（即使看起来用途不大）。

除去 nullptr 及 nullptr_t 以外，C++ 中还存在各种内置类型。C++11 标准严格规定了数据间的关系。大体上常见的规则简单地列在了下面：

- ❑所有定义为 nullptr_t 类型的数据都是等价的，行为也是完全一致。
- ❑nullptr_t 类型数据可以隐式转换成任意一个指针类型。
- ❑nullptr_t 类型数据不能转换为非指针类型，即使使用 reinterpret_cast<nullptr_t>() 的方式也是不可以的。
- ❑nullptr_t 类型数据不适用于算术运算表达式。
- ❑nullptr_t 类型数据可以用于关系运算表达式，但仅能与 nullptr_t 类型数据或者指针类型数据进行比较，当且仅当关系运算符为 ==、<=、>= 等时返回 true。

我们可以看看代码清单 7-3 所示的例子，这个例子集合了大多数我们需要的场景。

代码清单 7-3

```
#include <iostream>
#include <typeinfo>
using namespace std;

int main()
{
    // nullptr 可以隐式转换为 char*
    char * cp = nullptr;

    // 不可转换为整型，而任何类型也不能转换为 nullptr_t，
    // 以下代码不能通过编译
    // int n1 = nullptr;
    // int n2 = reinterpret_cast<int>(nullptr);

    // nullptr 与 nullptr_t 类型变量可以作比较，
    // 当使用 ==、<=、>= 符号比较时返回 true
    nullptr_t nptr;
    if (nptr == nullptr)
        cout << "nullptr_t nptr == nullptr" << endl;
```

```
    else
        cout << "nullptr_t nptr != nullptr" << endl;

    if (nptr < nullptr)
        cout << "nullptr_t nptr < nullptr" << endl;
    else
        cout << "nullptr_t nptr !< nullptr" << endl;

    // 不能转换为整型或 bool 类型，以下代码不能通过编译
    // if (0 == nullptr);
    // if (nullptr);

    // 不可以进行算术运算，以下代码不能通过编译
    // nullptr += 1;
    // nullprt * 5;

    // 以下操作均可以正常进行
    sizeof(nullptr);
    typeid(nullptr);
    throw(nullptr);

    return 0;
}

// 编译选项 :g++ 7-1-3.cpp -std=c++11
```

编译运行代码清单 7-3，我们可以得到以下结果：

```
nullptr_t nptr == nullptr
nullptr_t nptr !< nullptr
terminate called after throwing an instance of 'decltype(nullptr)'
Aborted
```

读者可以对应之前的规则试着分析一下上面的代码为什么有的不能够通过编译，以及为什么会产生上述的运行结果。

注意 如果读者的编译器能够编译 if(nullptr) 或者 if(nullptr == 0) 这样的语句，可能是因为编译器版本还不够新。老的 nullptr 定义中允许 nullptr 向 bool 的隐式转换，这带来了一些问题，而 C++11 标准中已经不允许这么做了。

此外，虽然 nullptr_t 看起来像是个指针类型，用起来更是，但在把 nullptr_t 应用于模板中时候，我们会发现模板却只能把它作为一个普通的类型来进行推导（并不会将其视为 T* 指针）。代码清单 7-4 所示的这个例子来源于 C++11 标准提案。

代码清单 7-4

```
#include <iostream>
```

```
using namespace std;

template<typename T> void g(T* t) {}
template<typename T> void h(T t) {}

int main()
{
   g(nullptr);              // 编译失败，nullptr 的类型是 nullptr_t，而不是指针
   g((float*) nullptr);     // 推导出 T = float

   h(0);                    // 推导出 T = int
   h(nullptr);              // 推导出 T = nullptr_t
   h((float*)nullptr);      // 推导出 T = float*
}

// 编译选项 :g++ 7-1-4.cpp -std=c++11
```

代码清单 7-4 中，g(nullptr) 并不会被编译器“智能”地推导成某种基本类型的指针（或者 void* 指针），因此要让编译器成功推导出 nullptr 的类型，必须做显式的类型转换。

7.1.3 一些关于 nullptr 规则的讨论

nullptr 这个名字看起来是比较古怪的。在 C++98 标准的时候，字符串 NULL 实际上已经得到了广泛的应用。而在 C++11 标准中，委员会采取了另起炉灶的方式，硬生生地添加了 nullptr 这个关键字，这让很多人表示不能理解。但另起炉灶而不是重用 NULL 的原因却是非常明显的。因为 NULL 已经是一个用途广泛的宏，且这个宏被不同的编译器实现为不同的解释，重用 NULL 会使得很多已有的 C++ 程序不能通过 C++11 编译器的编译。因此为了保证最大的兼容性，委员会采用了新的名称 nullptr，以避免和现有标识符的冲突。

此外在 C++11 标准中，nullptr 类型数据所占用的内存空间大小跟 void* 相同的，即：

```
sizeof(nullptr_t) == sizeof(void*)
```

关于这一点也可能引起疑惑，即是否 nullptr 就是 (void*)0 的一个别名。不过答案却是否定的，尽管两者看起来很相似，都可以被转换为任何类型的指针，但两者在语法层面有着不同的内涵。nullptr 是一个编译时期的常量，它的名字是一个编译时期的关键字，能够为编译器所识别。而 (void*)0 只是一个强制转换表达式，其返回的也是一个 void* 指针类型。

而且最为重要的是，在 C++ 语言中，nullptr 到任何指针的转换是隐式的，而 (void*)0 则必须经过类型转换后才能使用。我们可以看看代码清单 7-5 所示的例子。

代码清单 7-5

```
int foo()
{
    int* px = (void*)0;   // 编译错误，不能隐式地将无类型指针转换为 int* 类型的指针
```

```
    int* py = nullptr;
}
```

编译选项：g++ -std=c++11 7-1-5.cpp

可以看到，(void*)0 在使用上并不如 nullptr 方便。在 nullptr 出现之后，程序员大可以忘记 (void*)0，因为 nullptr 已经足够用了，而且也很好用。

注意 C 语言标准中的 void* 指针是可以隐式转换为任意指针的，这一点跟 C++ 是不同的。

此外，我们还注意到 C++11 标准有一条有趣的规定，nullptr_t 对象的地址可以被用户使用（虽然看起来好像没什么实用价值）。但这条规则有一点例外，就是虽然 nullptr 也是一个 nullptr_t 的对象，C++11 标准却规定用户不能获得 nullptr 的地址。其原因主要是因为 nullptr 被定义为一个右值常量，取其地址并没有意义。

不过 C++11 标准并没有禁止声明一个 nullptr 的右值引用，并打印其地址，因此我们在一些编译器上也做了个有趣的实验，对 nullptr 感兴趣的用户可以试运行一下代码清单 7-6 所示的这段的代码。

代码清单 7-6

```
#include <cstdio>
#include <cstddef>
using namespace std;

int main(){
    nullptr_t my_null;
    printf("%x\n", &my_null);

    // printf("%x", &nullptr);       // 根据 C++11 的标准设定，本句无法编译通过
    printf("%d\n", my_null == nullptr);

    const nullptr_t && default_nullptr = nullptr;      // default_nullptr 是
    nullptr 的一个右值引用
    printf("%x\n", &default_nullptr);
}

// 编译选项 :g++ -std=c++11 7-1-6.cpp
```

编译运行代码清单 7-6，我们的实验机上的结果看起来是这样：

```
7498fca8
1
7498fcb0
```

当然，运行结果跟所用编译器以及平台都有关系。不过，对于普通用户而言，需要记得的仅仅是，不要对 nullptr 做取地址操作即可。

7.2 默认函数的控制

☞ **类别：类作者**

7.2.1 类与默认函数

在 C++ 中声明自定义的类，编译器会默认帮助程序员生成一些他们未自定义的成员函数。这样的函数版本被称为“默认函数”。这包括了以下一些自定义类型的成员函数：

- ❑构造函数
- ❑拷贝构造函数
- ❑拷贝赋值函数（operator =）
- ❑移动构造函数
- ❑移动拷贝函数
- ❑析构函数

此外，C++ 编译器还会为以下这些自定义类型提供全局默认操作符函数：

- ❑operator,
- ❑operator &
- ❑operator &&
- ❑operator *
- ❑operator ->
- ❑operator ->*
- ❑operator new
- ❑operator delete

在 C++ 语言规则中，一旦程序员实现了这些函数的自定义版本，则编译器不会再为该类自动生成默认版本。有时这样的规则会被程序员忘记，最常见的是声明了带参数的构造版本，则必须声明不带参数的版本以完成无参的变量初始化。不过通过编译器的提示，这样的问题通常会得到更正。但更为严重的问题是，一旦声明了自定义版本的构造函数，则有可能导致我们定义的类型不再是 POD 的。我们可以看看代码清单 7-7 所示的例子。

代码清单 7-7

```
#include <type_traits>
#include <iostream>
using namespace std;
```

```
class TwoCstor {
public:
    // 提供了带参数版本的构造函数，则必须自行提供
    // 不带参数版本，且 TwoCstor 不再是 POD 类型
    TwoCstor() {};
    TwoCstor(int i): data(i) {}

private:
    int data;
};

int main(){
    cout << is_pod<TwoCstor>::value << endl;    // 0
}

// 编译选项 :g++ -std=c++11 7-2-1.cpp
```

代码清单 7-7 所示的例子中，程序员虽然提供了 TwoCstor() 构造函数，它与默认的构造函数接口和使用方式也完全一致，不过按照 3.6 节我们对“平凡的构造函数”的定义，该构造函数却不是平凡的，因此 TwoCstor 也就不再是 POD 的了。使用 is_pod 模板类查看 TwoCstor，也会发现程序输出为 0。对于形如 TwoCstor 这样只是想增加一些构造方式的简单类型而言，变为非 POD 类型带来一系列负面影响有时是程序员所不希望的（读者可以回顾一下 3.6 节，很多时候，这意味着编译器失去了优化这样简单的数据类型的可能）。因此客观上我们需要一些方式来使得这样的简单类型“恢复”POD 的特质。

而在 C++11 中，标准是通过提供了新的机制来控制默认版本函数的生成来完成这个目标的。这个新机制重用了 default 关键字。程序员可以在默认函数定义或者声明时加上“= default”，从而显式地指示编译器生成该函数的默认版本。而如果指定产生默认版本后，程序员不再也不应该实现一份同名的函数。具体如代码清单 7-8 所示。

代码清单 7-8

```
#include <type_traits>
#include <iostream>
using namespace std;

class TwoCstor {
public:
    // 提供了带参数版本的构造函数，再指示编译器
    // 提供默认版本，则本自定义类型依然是 POD 类型
    TwoCstor() = default;
    TwoCstor(int i): data(i) {}

private:
    int data;
```

```
};

int main(){
    cout << is_pod<TwoCstor>::value << endl; // 1
}

// 编译选项 :g++ 7-2-2.cpp -std=c++11
```

编译运行代码清单 7-8，会得到结果 1。TwoCstor 还是一个 POD 的类型。

另一方面，程序员在一些情况下则希望能够限制一些默认函数的生成。最典型地，类的编写者有时需要禁止使用者使用拷贝构造函数，在 C++98 标准中，我们的做法是将拷贝构造函数声明为 private 的成员，并且不提供函数实现。这样一来，一旦有人试图（或者无意识）使用拷贝构造函数，编译器就会报错。

我们来看看代码清单 7-9 所示的例子。

代码清单 7-9

```
#include <type_traits>
#include <iostream>
using namespace std;

class NoCopyCstor {
public:
    NoCopyCstor() = default;

private:
    // 将拷贝构造函数声明为 private 成员并不提供实现
    // 可以有效阻止用户错用拷贝构造函数
    NoCopyCstor(const NoCopyCstor &);
};

int main(){
    NoCopyCstor a;
    NoCopyCstor b(a);    // 无法通过编译
}

// 编译选项 :g++ 7-2-3.cpp -std=c++11
```

代码清单 7-9 中，NoCopyCstor b(a) 试图调用 private 的拷贝构造函数，该句编译不会通过。不过这样的做法也会对友元类或函数使用造成麻烦。友元类很可能需要拷贝构造函数，而简单声明 private 的拷贝构造函数不实现的话，会导致编译的失败。为了避免这种情况，我们还必须提供拷贝构造函数的实现版本，并将其声明为 private 成员，才能达到需要的效果。

在 C++11 中，标准则给出了更为简单的方法，即在函数的定义或者声明加上 “=

delete”。“= delete”会指示编译器不生成函数的缺省版本。我们可以看代码清单 7-10 所示的例子。

代码清单 7-10

```
#include <type_traits>
#include <iostream>
using namespace std;

class NoCopyCstor {
public:
    NoCopyCstor() = default;

    // 使用“= delete”同样可以有效阻止用户
    // 错用拷贝构造函数
    NoCopyCstor(const NoCopyCstor &) = delete;
};

int main(){
    NoCopyCstor a;
    NoCopyCstor b(a);    // 无法通过编译
}

// 编译选项 :g++ 7-2-4.cpp -std=c++11
```

代码清单 7-10 即是一个使用“= delete”删除拷贝构造函数的缺省版本的实例。值得注意的是，一旦缺省版本被删除了，重载该函数也是非法的。

7.2.2 “= default”与“= deleted”

在上面一节中，我们基本已经看到了 C++11 中“= default”和“= delete”的使用方法，事实上，C++11 标准称“= default”修饰的函数为显式缺省（explicit defaulted）函数，而称“= delete”修饰的函数为删除（deleted）函数。为了方便称呼，本书将删除函数称为显式删除函数。在下面的描述中，我们会沿用这些术语。

C++11 引入显式缺省和显式删除是为了增强对类默认函数的控制，让程序员能够更加精细地控制默认版本的函数。不过这并不是它们的唯一功能，而且使用上，也不仅仅局限在类的定义内。事实上，显式缺省不仅可以用于在类的定义中修饰成员函数，也可以在类定义之外修饰成员函数。代码清单 7-11 所示便是一个例子。

代码清单 7-11

```
class DefaultedOptr{
public:
    // 使用“= default”来产生缺省版本
    DefaultedOptr() = default;
```

```
    // 这里没使用“= default”
    DefaultedOptr & operator = (const DefaultedOptr & );
};

// 在类定义外用“= default”来指明使用缺省版本
inline DefaultedOptr &
DefaultedOptr::operator =( const DefaultedOptr & ) = default;

// 编译选项 :g++ -std=c++11 -c 7-2-5.cpp
```

在本例中，类 DefaultedOptr 的操作符 operator= 被声明在了类的定义外，并且被设定为缺省版本。这在 C++11 规则中也是被允许的。在类定义外显式指定缺省版本所带来的好处是，程序员可以对一个 class 定义提供多个实现版本。假设我们有下面的几个文件：

```
type.h : struct type { type(); };
type1.cc : type::type() = default;
type2.cc : type::type() { /*do some thing */ };
```

那么程序员就可以选择地编译 type1.cc 或者 type2.cc，从而轻易地在提供缺省函数的版本和使用自定义版本的函数间进行切换。这对于一些代码的调试是很有帮助的。

此外，除去我们在上节提到了多个可以由编译器默认提供的缺省函数，显式缺省还可以修饰一些其他函数，比如“operator ==”。C++11 标准并不要求编译器为这些函数提供缺省的实现，但如果将其声明为显式缺省的话，则编译器会按照某些“标准行为”为其生成所需要的版本。

关于显式删除，正如我们在上一节中看到，显式删除可以避免用户使用一些不应该使用的类的成员函数。不过显式删除也并非局限于成员函数，使用显式删除还可以避免编译器做一些不必要的隐式数据类型转换。我们来看看代码清单 7-12 所示的例子。

代码清单 7-12

```
class ConvType {
public:
    ConvType(int i) {};
    ConvType(char c) = delete;  // 删除 char 版本
};

void Func(ConvType ct) {}

int main() {
    Func(3);
    Func('a');  // 无法通过编译

    ConvType ci(3);
    ConvType cc('a');   // 无法通过编译
```

```
}

// 编译选项 :g++ -std=c++11 7-2-6.cpp
```

代码清单 7-12 中，我们显式删除了 ConvType(char) 版本的构造函数。则在调用 Func('a') 及构造变量 cc 的时候，编译器会给出错误提示并停止编译。这是因为编译器发现从 char 构造 ConvType 的方式是不被允许的。不过如果读者将 ConvType(char c) = delete; 这一句注释掉，代码清单 7-12 就可以通过编译了。这种情况下，编译器会隐式地将 a 转换为整型，并调用整型版本的构造函数。这样一来，我们就可以对一些危险的、不应该发生的隐式类型转换进行适当的控制。

不过我们还需要注意一下 explicit 关键字在这里可能产生的影响。让我们稍稍改变一下代码清单 7-12 中的代码，一些意想不到的结果就可能发生，如代码清单 7-13 所示。

代码清单 7-13

```
class ConvType {
public:
    ConvType(int i) {};
    explicit ConvType(char c) = delete;   // 删除 explicit 的 char 构造函数
};

void Func(ConvType ct) {}

int main() {
    Func(3);
    Func('a');        // 可以通过编译

    ConvType ci(3);
    ConvType cc('a');    // 无法通过编译
}

// 编译选项 :g++ -std=c++11 7-2-7.cpp
```

代码清单 7-13 中，语句 explicit ConvType(char) = delete 将从 char explicit 构造 ConvType 的方式显式删除了，这导致 cc 变量的构造不成功，因为其是显式构造的。不过在函数 Func 的调用中，编译器会尝试隐式地将 c 转换成 int，从而 Func('a') 的调用会导致一次 ConvType(int) 构造，因而能够通过编译。这样一来，explicit 带来了令人尴尬的效果，即没有彻底地禁止类型转换的发生。如果程序员发生了这样的错误，也可能会比较难找到原因。

事实上，在 C++11 提案中，提案的作者并不建议用户将 explicit 关键字和显式删除合用。因为两者的合用只会引起一些混乱性，并无什么好处。因此，程序员在使用显式删除时候，应该总是避免 explicit 关键字修饰的函数，反之亦然。

还有一点必须指出，对于使用显式删除来禁止编译器做一些不必要的类型转换上，我们

并不局限于缺省版本的类成员函数或者全局函数上，对于一些普通的函数，我们依然可以通过显式删除来禁止类型转换，如代码清单 7-14 所示。

代码清单 7-14

```
void Func(int i){};
void Func(char c) = delete;  // 显式删除 char 版本

int main(){
    Func(3);
    Func('c');  // 本句无法通过编译
    return 1;
}

// 编译选项 :g++ -std=c++11 7-2-8.cpp
```

代码清单 7-14 所示的例子中，我们显式删除了 Func 的 char 版本，这就会导致 Func('c') 调用的编译失败。

显式删除还有一些有趣的使用方式。比如程序员使用显式删除来删除自定义类型的 operator new 操作符的话，就可以做到避免在堆上分配该 class 的对象。代码清单 7-15 就是这样一个例子。

代码清单 7-15

```
#include <cstddef>

class NoHeapAlloc{
public:
    void * operator new(std::size_t) = delete;
};

int main(){
    NoHeapAlloc nha;
    NoHeapAlloc * pnha = new NoHeapAlloc;    // 编译失败
    return 1;
}

// 编译选项 :g++ -std=c++11 7-2-9.cpp
```

而在一些情况下，比如在代码清单 7-16 中，我们需要对象在指定内存位置进行内存分配，并且不需要析构函数来完成一些对象级别的清理。这个时候，我们可以通过显式删除析构函数来限制自定义类型在栈上或者静态的构造。

代码清单 7-16

```
#include <cstddef>
```

```
#include <new>

extern void* p;

class NoStackAlloc{
public:
    ~NoStackAlloc() = delete;
};

int main(){
    NoStackAlloc nsa;    // 无法通过编译
    new (p) NoStackAlloc(); // placement new, 假设 p 无需调用析构函数
    return 1;
}

// 编译选项 :g++ 7-2-10.cpp -std=c++11 -c
```

由于 placement new 构造的对象，编译器不会为其调用析构函数，因此析构函数被删除的类能够正常地构造。事实上，读者可以推而广之，将显式删除析构函数用于构建单件模式（Singleton），不过本书就不再展开讲了。

7.3 lambda 函数

☞ 类别：所有人

7.3.1 lambda 的一些历史

lambda（λ）在希腊字母表中位于第 11 位。同时，由于希腊数字是基于希腊字母的，所以 λ 在希腊数字中也表示了值 30。在数理逻辑或计算机科学领域中，lambda 则是被用来表示一种匿名函数，这种匿名函数代表了一种所谓的 λ 演算（lambda calculus）。

λ 演算是计算机语言领域的老古董，或者更确切地讲，λ 演算应该算做编程语言理论的研究成果，它的出现指引了实际编程语言的诞生。20 世纪 30 年代，阿隆佐·邱奇[⊖]（Alonzo Church）引入了这套表示计算（computation）的形式系统。1958 年，当时身在 MIT 的约翰·麦肯锡（John McCarthy）创造出了基于 λ 演算的 LISP 语言（但他并没有实现 LISP。第一个 Lisp 语言的实现是 Steve Russell 在 IBM 704 机器上完成的）。LISP 语言历史远早于 C 语言，可以算作第二古老的高级编程语言（第一个成功的高级编程语言是 IBM 的 FORTRAN），也是在学术界产生的第一个成功的编程语言，其被广泛应用于人工智能的研究领域。相比于基于 lambda 的 LISP 的成功，C++ 则显得非常年轻。直到 30 年后，Bjarne Stroustrup 才在贝尔实验室里开始设计并实现 C++。

⊖ 阿隆佐·邱奇是图灵的老师。

而从软件开发的角度看，以 lambda 概念为基础的“函数式编程”（Functional Programming）是与命令式编程（Imperative Programming）、面向对象编程（Object-orientated Programming）等并列的一种编程范型（Programming Paradigm）。现在的高级语言也越来越多地引入了多范型支持，很多近年流行的语言都提供了 lambda 的支持，比如 C#、PHP、JavaScript 等。而现在 C++11 也开始支持 lambda，并可能在标准演进过程中不停地进行修正。这样一来，从最早基于命令式编程范型的语言 C，到加入了面向对象编程范型血统的 C++，再到逐渐融入函数式编程范型的 lambda 的新语言规范 C++11，C/C++ 的发展也在融入多范型支持的潮流中。

7.3.2 C++11 中的 lambda 函数

lambda 的历时悠久，不过具体到 C++11 中，lambda 函数却显得与之前 C++ 规范下的代码在风格上有较大的区别。我们可以通过一个例子先来观察一下，如代码清单 7-17 所示。

代码清单 7-17

```
int main() {
    int girls = 3, boys = 4;
    auto totalChild = [](int x, int y)->int{ return x + y; };
    return totalChild(girls, boys);
}

编译选项：g++ -std=c++11 7-3-1.cpp
```

在代码清单 7-17 所示的例子当中，我们定义了一个 lambda 函数。该函数接受两个参数 (int x, int y)，并且返回其和。直观地看，lambda 函数跟普通函数相比不需要定义函数名，取而代之的多了一对方括号（[]）。此外，lambda 函数还采用了追踪返回类型的方式声明其返回值。其余方面看起来则跟普通函数定义一样。

而通常情况下，lambda 函数的语法定义如下：

```
[capture](parameters) mutable ->return-type{statement}
```

其中，

- [capture]：捕捉列表。捕捉列表总是出现在 lambda 函数的开始处。事实上，[] 是 lambda 引出符。编译器根据该引出符判断接下来的代码是否是 lambda 函数。捕捉列表能够捕捉上下文中的变量以供 lambda 函数使用。具体的方法在下文中会再描述。
- (parameters)：参数列表。与普通函数的参数列表一致。如果不需要参数传递，则可以连同括号 () 一起省略。
- mutable：mutable 修饰符。默认情况下，lambda 函数总是一个 const 函数，mutable 可以取消其常量性。在使用该修饰符时，参数列表不可省略（即使参数为空）。
- ->return-type：返回类型。用追踪返回类型形式声明函数的返回类型。出于方便，不

需要返回值的时候也可以连同符号 -> 一起省略。此外，在返回类型明确的情况下，也可以省略该部分，让编译器对返回类型进行推导。

- {statement}：函数体。内容与普通函数一样，不过除了可以使用参数之外，还可以使用所有捕获的变量。

在 lambda 函数的定义中，参数列表和返还类型都是可选的部分，而捕捉列表和函数体都可能为空。那么在极端情况下，C++11 中最为简略的 lambda 函数只需要声明为

```
[]{};
```

就可以了。不过理所应当地，该 lambda 函数不能做任何事情。

代码清单 7-18 中列出了各种各样的 lambda 函数。

代码清单 7-18

```
int main(){
    []{};                    // 最简 lambda 函数
    int a = 3;
    int b = 4;
    [=] { return a + b;};          // 省略了参数列表与返回类型，返回类型由编译器推断为 int
    auto fun1 = [&](int c) { b = a + c; };       // 省略了返回类型，无返回值
    auto fun2 = [=, &b](int c)->int { return b += a + c; };// 各部分都很完整的 lambda 函数
}

// 编译选项 :g++ -std=c++11 7-3-2.cpp
```

在代码清单 7-18 中，我们看到了各种各样的捕捉列表的使用。直观地讲，lambda 函数与普通函数可见的最大区别之一，就是 lambda 函数可以通过捕捉列表访问一些上下文中的数据。具体地，捕捉列表描述了上下文中哪些的数据可以被 lambda 使用，以及使用方式（以值传递的方式或引用传递的方式）。在代码清单 7-17 的例子中，我们是使用参数的方式传递变量，现在让我们使用捕捉列表来改写这个例子，如代码清单 7-19 所示。

代码清单 7-19

```
int main() {
    int boys = 4, int girls = 3;
    auto totalChild = [girls, &boys]()->int{ return girls + boys; };
    return totalChild();
}

// 编译选项 :: g++ -std=c++11 7-3-3.cpp
```

代码清单 7-19 中，我们使用了捕捉列表捕捉上下文中的变量 girls、boys。与代码清单 7-17 相比，函数的原型发生了变化，即 totalChild 不再需要传递参数。这个改变看起来平淡无奇，不过读者在阅读 7.3.3 节之后可以知道，此时 girls 和 boys 可以视为 lambda 函数的一

种初始状态，lambda 函数的运算则是基于初始状态进行的运算。这与函数简单基于参数的运算是有所不同的。

语法上，捕捉列表由多个捕捉项组成，并以逗号分割。捕捉列表有如下几种形式：

- ❑[var] 表示值传递方式捕捉变量 var。
- ❑[=] 表示值传递方式捕捉所有父作用域的变量（包括 this）。
- ❑[&var] 表示引用传递捕捉变量 var。
- ❑[&] 表示引用传递捕捉所有父作用域的变量（包括 this）。
- ❑[this] 表示值传递方式捕捉当前的 this 指针。

注意 父作用域：enclosing scope，这里指的是包含 lambda 函数的语句块，在代码清单 7-19 中，即 main 函数的作用域。

通过一些组合，捕捉列表可以表示更复杂的意思。比如：

- ❑[=, &a, &b] 表示以引用传递的方式捕捉变量 a 和 b，值传递方式捕捉其他所有变量。
- ❑[&, a, this] 表示以值传递的方式捕捉变量 a 和 this，引用传递方式捕捉其他所有变量。

不过值得注意的是，捕捉列表不允许变量重复传递。下面一些例子就是典型的重复，会导致编译时期的错误。

- ❑[=, a] 这里 = 已经以值传递方式捕捉了所有变量，捕捉 a 重复。
- ❑[&, &this] 这里 & 已经以引用传递方式捕捉了所有变量，再捕捉 this 也是一种重复。

利用以上的规则，对于代码清单 7-19 的 lambda 函数，我们可以通过 [=] 来声明捕捉列表，进而对 totalChild 书写上的进一步简化，如代码清单 7-20 所示。

代码清单 7-20

```
int main() {
    int boys = 4, girls = 3;
    auto totalChild = [=]()->int{ return girls + boys; };// 捕捉所有父作用域的变量
    return totalChild();
}

// 编译选项 :: g++ -std=c++11 7-3-4.cpp
```

通过捕捉列表 [=]，lambda 函数的父作用域中所有自动变量都被 lambda 依照传值的方式捕捉了。

必须指出的是，依照现行 C++11 标准，在块作用域（block scope，可以简单理解为在 {} 之内的任何代码都是块作用域的）以外的 lambda 函数捕捉列表必须为空。因此这样的 lambda 函数除去语法上的不同以外，跟普通函数区别不大。而在块作用域中的 lambda 函数

仅能捕捉父作用域中的自动变量，捕捉任何非此作用域或者是非自动变量（如静态变量等）都会导致编译器报错。

7.3.3 lambda 与仿函数

好的编程语言一般都有好的库支持，C++ 也不例外。C++ 语言在标准程序库 STL 中向用户提供了一些基本的数据结构及一些基本的算法等。在 C++11 之前，我们在使用 STL 算法时，通常会使用到一种特别的对象，一般来说，我们称之为函数对象，或者仿函数（functor）。仿函数简单地说，就是重定义了成员函数 operator () 的一种自定义类型对象。这样的对象有个特点，就是其使用在代码层面感觉跟函数的使用并无二样，但究其本质却并非函数。我们可以看一个仿函数的例子，如代码清单 7-21 所示。

代码清单 7-21

```
class _functor {
public:
    int operator()(int x, int y) { return x + y; }
};

int main(){
    int girls = 3, boys = 4;
    _functor totalChild;
    return totalChild(5, 6);
}

// 编译选项 :g++ 7-3-5.cpp -std=c++11
```

这个例子中，class _functor 的 operator() 被重载，因此，在调用该函数的时候，我们看到跟函数调用一样的形式，只不过这里的 totalChild 不是函数名称，而是对象名称。

注意 相比于函数，仿函数可以拥有初始状态，一般通过 class 定义私有成员，并在声明对象的时候对其进行初始化。私有成员的状态就成了仿函数的初始状态。而由于声明一个仿函数对象可以拥有多个不同初始状态的实例，因此可以借由仿函数产生多个功能类似却不同的仿函数实例（这里是一个多状态的仿函数的实例）。

```
#include <iostream>
using namespace std;

class Tax {
private:
    float rate;
    int base;
public:
    Tax(float r, int b): rate(r), base(b){}
```

```
    float operator() (float money) { return (money - base) * rate; }
};

int main() {
    Tax high(0.40, 30000);
    Tax middle(0.25, 20000);
    cout << "tax over 3w: " << high(37500) << endl;
    cout << "tax over 2w: " << middle(27500) << endl;
    return 0;
}
```

这里通过带状态的仿函数，可以设定两种不同的税率的计算。

而仔细观察的话，除去自定义类型 _functor 的声明及其对象的定义，可以发现代码清单 7-21 跟代码清单 7-17 中 lambda 函数的定义看起非常类似。这是否说明仿函数跟 lambda 在实现之间存在着一种默契呢？我们可以再来看一个例子，如代码清单 7-22 所示。

代码清单 7-22

```
class AirportPrice{
private:
    float _dutyfreerate;
public:
    AirportPrice(float rate): _dutyfreerate(rate){}
    float operator()(float price) {
        return price * (1 - _dutyfreerate/100);
    }
};

int main(){
    float tax_rate = 5.5f;
    AirportPrice Changi(tax_rate);

    auto Changi2 =
        [tax_rate](float price)->float{ return price * (1 - tax_rate/100); };

    float purchased = Changi(3699);

    float purchased2 = Changi2(2899);
}

// 编译选项 :g++ 7-3-6.cpp -std=c++11
```

代码清单 7-22 是一个机场返税的例子。该例中，分别使用了仿函数和 lambda 两种方式来完成扣税后的产品价格计算。在这里我们看到，lambda 函数捕捉了 tax_rate 变量，而仿函数则以 tax_rate 初始化类。其他的，如在参数传递上，两者保持一致。可以看到，除去在语法层面上的不同，lambda 和仿函数却有着相同的内涵——都可以捕捉一些变量作为初始状态，并接受参数进行运算。

而事实上，仿函数是编译器实现 lambda 的一种方式。在现阶段，通常编译器都会把 lambda 函数转化为成为一个仿函数对象。因此，在 C++11 中，lambda 可以视为仿函数的一种等价形式了，或者更动听地说，lambda 是仿函数的“语法甜点”。

我们可以通过图 7-1 展现代码清单 7-22 中的 lambda 函数和仿函数是如何等价的。

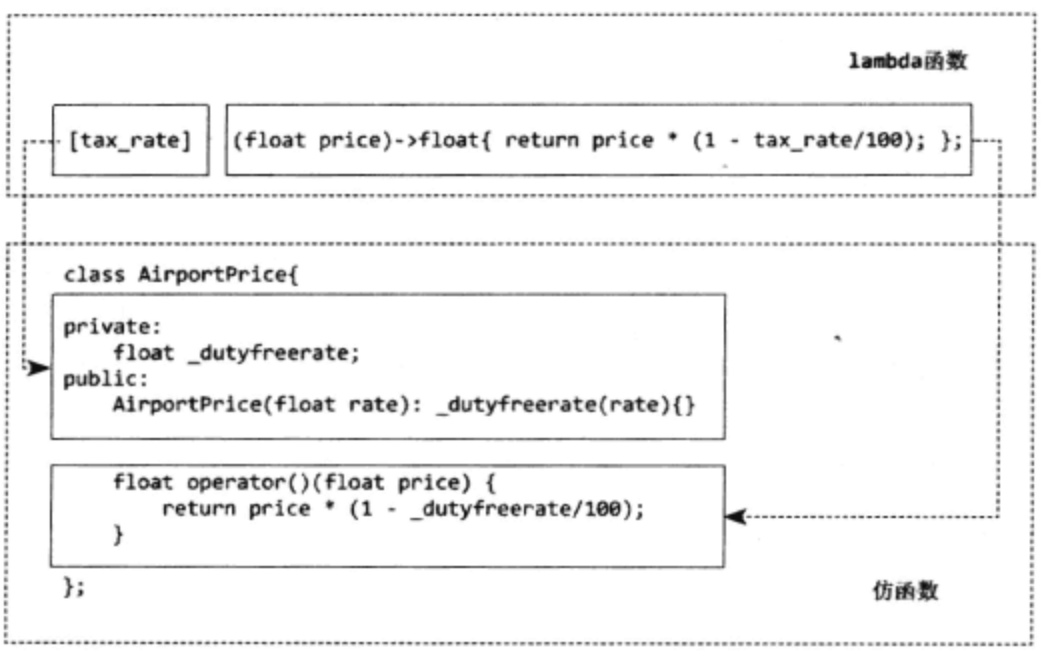


图 7-1 lambda 函数及与其等价的仿函数

注意 有的时候，我们在编译时发现 lambda 函数出现了错误，编译器会提示一些构造函数等相关信息。这显然是由于 lambda 的这种实现方式造成的。理解了这种实现，用户也就能够正确理解错误信息的由来。

如前面提到的，仿函数被广泛地用于 STL 中，同样的，在 C++11 中，lambda 也在标准库中被广泛地使用。由于其书写简单，通常可以就地定义，因此用户常可以使用 lambda 代替仿函数来书写代码，我们可以在 7.3.4 节中看到相关的应用方式，在 7.3.6 中了解 lambda 何时可以取代仿函数。

7.3.4 lambda 的基础使用

依据 lambda 的语法，编写 lambda 函数是非常容易的。不过用得上 lambda 函数的地方比较特殊。最为简单的应用下，我们会利用 lambda 函数来封装一些代码逻辑，使其不仅具有函数的包装性，也具有就地可见的自说明性。让我们首先来看一个例子，如代码清单 7-23 所示。

代码清单 7-23

```
extern int z;
```

```
extern float c;
void Calc(int& , int, float &, float);

void TestCalc() {
    int x, y = 3;
    float a, b = 4.0;

    int success = 0;

    auto validate = [&]() -> bool
    {
        if ((x == y + z) && (a == b + c))
            return 1;
        else
            return 0;
    };

    Calc(x, y, a, b);
    success += validate();

    y = 1024;
    b = 1e13;

    Calc(x, y, a, b);
    success += validate();
}
```

```
// 编译选项 :g++ -c -std=c++11 7-3-7.cpp
```

在代码清单 7-23 所示的例子中，用户试图用自己写的函数 TestCalc 进行测试。这里使用了一个 auto 关键字推导出了 validate 变量的类型为匿名 lambda 函数。可以看到，我们使用 lambda 函数直接访问了 TestCal 中的局部的变量来完成这个工作。

在没有 lambda 函数之前，通常需要在 TestCalc 外声明同样一个函数，并且把 TestCalc 中的变量当作参数进行传递。出于函数作用域及运行效率考虑，这样声明的函数通常还需要加上关键字 static 和 inline。相比于一个传统意义上的函数定义，lambda 函数在这里更加直观，使用起来也非常简便，代码可读性很好，效果上，lambda 函数则等同于一个“局部函数”。

注意 局部函数（local function，即在函数作用域中定义的函数），也称为内嵌函数（nested function）。局部函数通常仅属于其父作用域，能够访问父作用域的变量，且在其父作用域中使用。C/C++ 语言标准中不允许局部函数存在（不过一些其他语言是允许的，比如 FORTRAN），C++11 标准却用比较优雅的方式打破了这个规则。因为事实上，lambda 可以像局部函数一样使用。

必须指出的是，相比于在函数外定义的 static inline 函数，或者是自定义的宏，本例中 lambda 函数并没有实际运行时的性能优势（但也不会差，lambda 函数在 C++11 标准中默认是内联的）。同局部函数一样，lambda 函数在代码的作用域上仅属于其父作用域，不过直观地看，lambda 函数代码的可读性可能更好，尤其对于小的函数而言。

对于运算比较复杂的函数（比如实现一些很复杂的算法），通常函数中会有大量的局部状态（变量），这个时候如果程序员只是需要一些“局部”的功能——比如打印一些内部状态，或者做一些固定的操作，这些功能往往不能与其他任何的代码共享，却要在一个函数中多次重用。那么使用 lambda 的捕捉列表功能则相较于独立的全局静态函数或私有成员函数方便很多。设计一个仅使用捕捉列表 lambda 函数不会像设计传统函数一样要关心大量细节：需要多少参数、哪些参数需要按值传递、哪些参数需要按引用或者指针传递，通通不需要程序员来考虑。而且父函数结束后，该 lambda 函数也就不再可用了，不会污染任何名字空间（当然，事实上如我们所讲的，lambda 本身就是匿名的函数）。因此，对于复杂代码的快速开发而言，lambda 的出现意义重大。事实上，这些也是局部函数的好处。在 C++11 之前，程序员只能通过编写类来模拟局部函数，实现复杂而且常常会存在着各种各样的问题，lambda 的出现使得这样的做法统统成为了历史。

我们再来看一个常量性的例子。在编写程序的时候，程序员通常会发现自己需要一些“常量”，不过这些常量的值却由自己初始化状态决定的。我们来看看代码清单 7-24 所示的例子。

代码清单 7-24

```
int Prioritize(int );

int AllWorks(int times){
    int i;

    int x;
    try {
        for (i = 0; i < times; i++)
            x += Prioritize(i);
    }
    catch(...) {
        x = 0;
    }

    const int y = [=]{
        int i, val;
        try {
            for (i = 0; i < times; i++)
                val += Prioritize(i);
        }
        catch(...) {
```

```
            val = 0;
        }
        return val;
    }();
}

// 编译选项 :g++ -std=c++11 7-3-8.cpp -c
```

在代码清单 7-24 所示的例子中，我们对 x 和 y 的初始化实际是完全一致的。可以看到，x（或 y）的初始化需要循环调用函数 Prioritize，并且在 Prioritize 抛出异常的时候对 x（或 y）赋默认值 0。

在不使用函数的情况下，由于初始化要在运行时修改 x 的值，因此，虽然 x 在初始化之后对于程序而言是个常量，却不能被声明为 const。而在定义 y 的时候，由于我们就地定义 lambda 函数并且调用，y 仅需使用其返回值，于是常量性得到了保证。

读者可能觉得也可以定义一个普通函数来完成 y 的常量定义，但对于代码清单 7-24 中的代码量较少，自说明意义较强的初始化而言，无疑采用 lambda 函数初始化的时候，代码的可读性会更好。而且这么写，我们也就不需要为这段代码逻辑取个函数名（通常 init 这样的名字是正确的，但是却很难解释清楚代码做了什么）。这是 lambda 函数的一个优势所在。

7.3.5 关于 lambda 的一些问题及有趣的实验

使用 lambda 函数的时候，捕捉列表不同会导致不同的结果。具体地讲，按值方式传递捕捉列表和按引用方式传递捕捉列表效果是不一样的。对于按值方式传递的捕捉列表，其传递的值在 lambda 函数定义的时候就已经决定了。而按引用传递的捕捉列表变量，其传递的值则等于 lambda 函数调用时的值。我们可以看一下代码清单 7-25 所示的例子。

代码清单 7-25

```
#include <iostream>
using namespace std;

int main() {
    int j = 12;
    auto by_val_lambda = [=] { return j + 1;};
    auto by_ref_lambda = [&] { return j + 1;};
    cout << "by_val_lambda: " << by_val_lambda() << endl;
    cout << "by_ref_lambda: " << by_ref_lambda() << endl;

    j++;
    cout << "by_val_lambda: " << by_val_lambda() << endl;
    cout << "by_ref_lambda: " << by_ref_lambda() << endl;
}
```

编译选项：g++ -std=c++11 7-3-9.cpp

完成编译运行后，我们可以看到运行结果如下：

```
by_val_lambda: 13
by_ref_lambda: 13
by_val_lambda: 13
by_ref_lambda: 14
```

第一次调用 by_val_lambda 和 by_ref_lambda 时，其运算结果并没有不同。两者均计算的是 12+1=13。但在第二次调用 by_val_lambda 的时候，其计算的是 12+1=13，相对地，第二次调用 by_ref_lambda 时计算的是 13+1=14。这个结果的原因是由于在 by_val_lambda 中，j 被视为了一个常量，一旦初始化后不会再改变（可以认为之后只是一个跟父作用域中 j 同名的常量）而在 by_ref_lambda 中，j 仍在使用父作用域中的值。

因此简单地总结的话，在使用 lambda 函数的时候，如果需要捕捉的值成为 lambda 函数的常量，我们通常会使用按值传递的方式捕捉；反之，需要捕捉的值成为 lambda 函数运行时的变量（类似于参数的效果），则应该采用按引用方式进行捕捉。

此外，关于 lambda 函数的类型以及该类型跟函数指针之间的关系，读者可能存在一些困惑。回顾之前的例子，大多数情况下把匿名的 lambda 函数赋值给了一个 auto 类型的变量，这是一种声明和使用 lambda 函数的方法。结合之前关于 auto 的知识，有人会猜测 totalChild 是一种函数指针类型的变量，而在阅读过 7.3.2 节关于 lambda 与仿函数之间关系之后，大多数读者会更倾向于认为 lambda 是一种自定义类型。而事实上，lambda 的类型并非简单的函数指针类型或者自定义类型。

从 C++11 标准的定义上可以发现，lambda 的类型被定义为“闭包”（closure）的类[⊖]，而每个 lambda 表达式则会产生一个闭包类型的临时对象（右值）。因此，严格地讲，lambda 函数并非函数指针。不过 C++11 标准却允许 lambda 表达是向函数指针的转换，但前提是 lambda 函数没有捕捉任何变量，且函数指针所示的函数原型，必须跟 lambda 函数有着相同的调用方式。我们可以通过代码清单 7-26 所示的这个例子来说明。

代码清单 7-26

```
int main() {
    int girls = 3, boys = 4;
    auto totalChild = [](int x, int y)->int{ return x + y; };
    typedef int (*allChild)(int x, int y);
    typedef int (*oneChild)(int x);

    allChild p;
```

⊖ C++11 标准定义，closure 类型被定义为特有的（unique）、匿名且非联合体（unnamed nonunion）的 class 类型。

```
    p = totalChild;

    oneChild q;
    q = totalChild;      // 编译失败，参数必须一致

    decltype(totalChild) allPeople = totalChild;    // 需通过 decltype 获得 lambda 的类型
    decltype(totalChild) totalPeople = p;       // 编译失败，指针无法转换为 lambda
    return 0;
}

// 编译选项 :g++ -std=c++11 7-3-10.cpp
```

在代码清单 7-26 所示的例子中，我们可以把没有捕捉列表的 totalChild 转化为接受参数类型相同的 allChild 类型的函数指针。不过，转化为参数类型不一致的 oneChild 类型则会失败。此外，将函数指针转化为 lambda 也是不成功的（虽然似乎 C++11 标准并没有明确禁止这一点）。

值得注意的是，程序员也可以通过 decltype 的方式来获得 lambda 函数的类型。其方式如同代码清单 7-26 中声明 allPeople 一样，虽然使用 decltype 来获得 lambda 函数类型的做法不是很常见，但在实例化一些模板的时候使用该方法会较为有用。

除此之外，还有一个问题是关于 lambda 函数的常量性及 mutable 关键字的。我们来看看代码清单 7-27 所示的这个例子[⊖]。

代码清单 7-27

```
int main(){
    int val;
    // 编译失败，在 const 的 lambda 中修改常量
    auto const_val_lambda = [=]() { val = 3;};

    // 非 const 的 lambda，可以修改常量数据
    auto mutable_val_lambda = [=]() mutable { val = 3;};

    // 依然是 const 的 lambda，不过没有改动引用本身
    auto const_ref_lambda = [&] { val = 3;};

    // 依然是 const 的 lambda，通过参数传递 val
    auto const_param_lambda = [&](int v) { v = 3;};
    const_param_lambda(val);

    return 0;
}

// 编译选项 :g++ -std=c++11 7-3-11.cpp
```

⊖ 该例子的问题实际上来源自网络上的一次讨论：http://stackoverflow.com/questions/5501959/why-does-c0xs-lambda-require-mutable-keyword-for-capture-by-value-by-defau。

在代码清单 7-27 所示的例子中，我们定义了 4 种不同的 lambda 函数，这 4 种 lambda 函数本身的行为都是一致的，即修改父作用域中传递而来的 val 参数的值。不过对于 const_val_lambda 函数而言，编译器认为这是一个错误。

```
7-3-10.cpp: In lambda function:
7-3-10.cpp:4:43: error: assignment of read-only variable 'val'
```

而对于声明了 mutable 属性的 mutable_val_lambda 函数，以及通过引用传递变量 val 的 const_ref_lambda 函数，甚至是通过参数来传递变量 val 的 const_param_lambda，编译器均不会报错。

如我们之前的定义中提到一样，C++11 中，默认情况下 lambda 函数是一个 const 函数。按照规则，一个 const 的成员函数是不能在函数体中改变非静态成员变量的值的。但这里明显编译器对不同传参或捕捉列表的 lambda 函数执行了不同的规则有着不同的见解。其究竟是基于什么样的规则而做出了这样的决定呢?

初看这个问题比较让人困惑，但事实上这跟 lambda 函数的特别的常量性相关。这里我们还是需要使用 7.3.3 中的知识将 lambda 函数转化为一个完整的仿函数，需要注意的是，lambda 函数的函数体部分，被转化为仿函数之后会成为一个 class 的常量成员函数。整个 const_val_lambda 看起来会是代码清单 7-28 所示代码的样子。

代码清单 7-28

```
class const_val_lambda{
public:
    const_val_lambda(int v): val(v){}

public:
    void operator()() const { val = 3; }/* 注意：常量成员函数 */

private:
    int val;
};

// 编译选项 :g++ -std=c++11 7-3-12.cpp -c
```

对于常量成员函数，其常量的规则跟普通的常量函数是不同的。具体而言，对于常量成员函数，不能在函数体内改变 class 中任何成员变量。因此，如果将代码清单 7-28 中的仿函数替代代码清单 7-27 中的 lambda 函数，编译报错则显得理所应当。

现在问题就比较清楚了。lambda 的捕捉列表中的变量都会成为等价仿函数的成员变量（如 const_val_lambda 中的成员 val），而常量成员函数（如 operator()）中改变其值是不允许的，因而按值捕捉的变量在没有声明为 mutable 的 lambda 函数中，其值一旦被修改就会导致编译器报错。

而使用引用的方式传递的变量在常量成员函数中值被更改则不会导致错误。关于这一点在很多 C++ 书籍中已经有过讨论。简单地说，由于函数 const_ref_lambda 不会改变引用本身，而只会改变引用的值，因此编译器将编译通过。至于按传递参数的 const_param_lambda，就更加不会引起编译器的“抱怨”了。

准确地讲，现有 C++11 标准中的 lambda 等价的是有常量 operator() 的仿函数。因此在使用捕捉列表的时候必须注意，按值传递方式捕捉的变量是 lambda 函数中不可更改的常量（如同我们之前在按值和按引用方式捕捉的讨论中提到的一样，不过现在我们已经在语言层面看到了限制）。标准这么设计可能是源自早期 STL 算法一些设计上的缺陷（对仿函数没有做限制，从而导致一些设计不算特别良好的算法出错，可以参考 Scott Mayer 的 Effective STL item 39，或者 Nicolai M.Josuttis 的 The C++ Standard Library - A Tutorial and Reference 关于仿函数的部分）。而更一般地讲，这样的设计有其合理性，改变从上下文中拷贝而来的临时变量通常不具有任何意义。绝大多数时候，临时变量只是用于 lambda 函数的输入，如果需要输出结果到上下文，我们可以使用引用，或者通过让 lambda 函数返回值来实现。

此外，lambda 函数的 mutable 修饰符可以消除其常量性，不过这实际上只是提供了一种语法上的可能性，现实中应该没有多少需要使用 mutable 的 lambda 函数的地方。大多数时候，我们使用默认版本的（非 mutable）的 lambda 函数也就足够了。

注意 关于按值传递捕捉的变量不能被修改这一点，有人认为这算是“闭包”类型的名称的体现，即在复制了上下文中变量之后关闭了变量与上下文中变量的联系，变量只与 lambda 函数运算本身有关，不会影响 lambda 函数（闭包）之外的任何内容。

7.3.6 lambda 与 STL

如 7.3.3 节中讲到的，lambda 对 C++11 最大的贡献，或者说是改变，应该在 STL 库中。而更具体地说，我们会发现使用 STL 的算法更加容易了，也更加容易学习了。

首先我们来看一个最为常见的 STL 算法 for_each。简单地说，for_each 算法的原型如下：

```
UnaryProc for_each(InputIterator beg, InputIterator end, UnaryProc op)
```

让我们忽略一些细节，大概讲，for_each 算法需要一个标记开始的 iterator，一个标记结束的 iterator，以及一个接受单个参数的“函数”（即一个函数指针、仿函数或者 lambda 函数）。

for_each 的一个示意实现如下：

```
for_each(iterator begin, iterator end, Function fn) {
    for (iterator i = begin; i != end; ++i)   fn(*i);
}
```

通过 for_each，我们可以完成各种循环操作，如代码清单 7-29 所示。

代码清单 7-29

```
#include <vector>
#include <algorithm>
using namespace std;

vector<int> nums;
vector<int> largeNums;

const int ubound = 10;

inline void LargeNumsFunc(int i){
    if (i > ubound)
        largeNums.push_back(i);
}

void Above() {
    // 传统的 for 循环
    for (auto itr = nums.begin(); itr != nums.end(); ++itr) {
        if (*itr >= ubound)
            largeNums.push_back(*itr);
    }

    // 使用函数指针
    for_each(nums.begin(), nums.end(), LargeNumsFunc);

    // 使用 lambda 函数和算法 for_each
    for_each(nums.begin(), nums.end(), [=](int i){
            if (i > ubound)
                largeNums.push_back(i);
            });
}

编译选项：g++ 7-3-13.cpp -c -std=c++11
```

在代码清单 7-29 的例子当中，我们分别用了 3 种方式来遍历一个 vector nums，找出其中大于 ubound 的值，并将其写入另外一个 vector largeNums 中。第一种是传统的 for 循环；第二种，则更泛型地使用了 for_each 算法以及函数指针；第三种同样使用了 for_each，但是第三个参数传入的是 lambda 函数。

首先必须指出的是使用 for_each 的好处。如 Scott Mayer 在 Effective STL（item43）中提到的一样，使用 for_each 算法相较于手写的循环在效率、正确性、可维护性上都具有一定优势。最典型的，程序员不用关心 iterator，或者说循环的细节，只需要设定边界，作用于每个元素的操作，就可以在近似“一条语句”内完成循环，正如函数指针版本和 lambda 版本完成的那样。

那么我们再比较一下函数指针方式以及 lambda 方式。函数指针的方式看似简洁，不过却有很大的缺陷。第一点是函数定义在别的地方，比如很多行以前（后）或者别的文件中，这样的代码阅读起来并不方便。第二点则是出于效率考虑，使用函数指针很可能导致编译器不对其进行 inline 优化（inline 对编译器而言并非强制），在循环次数较多的时候，内联的 lambda 和没有能够内联的函数指针可能存在着巨大的性能差别。因此，相比于函数指针，lambda 拥有无可替代的优势。

此外，函数指针的应用范围相对狭小，尤其是我们需要具备一些运行时才能决定的状态的时候，函数指针就会捉襟见肘了。倘若回到 10 年前（C++98 时代），遇到这种情况的时候，迫切想应用泛型编程的 C++ 程序员或许会毫不犹豫地使用仿函数，不过现在我们则没有必要那么做。

我们稍微修改一下代码清单 7-29 所示的例子，如代码清单 7-30 所示。

代码清单 7-30

```
#include <vector>
#include <algorithm>
using namespace std;

vector<int> nums;
vector<int> largeNums;

class LNums{
    public:
        LNums(int u): ubound(u){}

        void operator () (int i) const
        {
            if (i > ubound)
                largeNums.push_back(i);
        }
    private:
        int ubound;
};

void Above(int ubound) {
    // 传统的 for 循环
    for (auto itr = nums.begin(); itr != nums.end(); ++itr) {
        if (*itr >= ubound)
            largeNums.push_back(*itr);
    }

    // 使用仿函数
    for_each(nums.begin(), nums.end(), LNums(ubound));

    // 使用 lambda 函数和算法 for_each
```

```
    for_each(nums.begin(), nums.end(), [=](int i){
            if (i > ubound)
                largeNums.push_back(i);
            });

}

// 编译选项:g++ 7-3-14.cpp -std=c++11 -c
```

在代码清单7-30中，为了函数的最大可用性，我们把代码清单7-29中的全局变量ubound变成了代码清单7-30中的Above的参数。这样一来，我们传递给for_each函数的第三个参数（函数指针，仿函数或是lambda）而言就需要知道ubound是多少。由于函数只能通过参数传递这个状态（ubound），那么除非for_each调用函数的方式做出改变（比如增加调用函数的参数），否则编译器不会让其通过编译。因此，这个时候拥有状态的仿函数是更佳的选择。不过比较上面的代码，我们可以直观地看到lambda函数比仿函数书写上的简便性。因此，lambda在这里依然是最佳的选择（如果读者觉得这样还不够过瘾的话，那可以尝试一下C++11新风格的for循环。对于代码清单7-30所示的这个例子，新风格的for会更加简练）。

注意 事实上，STL算法对传入的“函数”的原型有着严格的说明，像for_each就只能传入使用单参数进行调用的“函数”。有的时候用户可以通过STL的一些adapter来改变参数个数，不过必须了解的是，这些adapter其实也是仿函数。

在C++98时代，STL中其实也内置了一些仿函数供程序员使用。代码清单7-31所示的例子中，我们将重点比较一下这些内置仿函数与lambda使用上的特点。

代码清单7-31

```
#include <vector>
#include <algorithm>
using namespace std;

extern vector<int> nums;

void OneCond(int val){
   // 传统的for循环
   for (auto i = nums.begin(); i != nums.end(); i++)
      if (*i == val) break;

   // 内置的仿函数(template) equal_to, 通过bind2nd使其成为单参数调用的仿函数
   find_if(nums.begin(), nums.end(), bind2nd(equal_to<int>(), val));

   // 使用lambda函数
   find_if(nums.begin(), nums.end(), [=](int i) {
            return i == val;
```

```
            });
}

// 编译选项：g++ -c -std=c++11 7-3-15.cpp
```

在代码清单 7-31 中，我们还是列出了 3 种方式来寻找 vector nums 中第一个值等于 val 的元素。我们可以看一下使用内置仿函数的方式。没有太多接触过 STL 算法的人可能对 bind2nd(equal_to<int>(), val) 这段代码相当迷惑，但简单地说，就是定义了一个功能是比较 i==val 的仿函数，并通过一定方式（bind2nd）使其函数调用接口只需要一个参数即可。反观 lambda 函数，其意义简洁明了，使用者使用的时候，也不需要有太多的背景知识。

但现在为止，我们并不能说 lambda 已经赢过了内置仿函数。而实际上，内置的仿函数应用范围很受限制，我们改动一下代码清单 7-31，使其稍微复杂一点，如代码清单 7-32 所示。

代码清单 7-32

```
#include <vector>
#include <algorithm>
using namespace std;

extern vector<int> nums;

void TwoCond(int low, int high) {
    // 传统的 for 循环
    for (auto i = nums.begin(); i != nums.end(); i++)
        if (*i >= low && *i < high) break;

    // 利用了 3 个内置的仿函数，以及非标准的 compose2
    find_if(nums.begin(), nums.end(),
            compose2(logical_and<bool>(),
                bind2nd(less<int>(), high),
                bind2nd(greater_equal<int>(), low)));

    // 使用 lambda 函数
    find_if(nums.begin(), nums.end(), [=](int i) {
            return i >= low && i < high;
            });
}

// 编译选项：g++ -c -std=c++11 7-3-16.cpp
```

在代码清单 7-32 中，我们将代码清单 7-31 中的判断条件稍微调整得复杂了一些，即需找到 vector nums 中第一个值介于 [low, high) 间的元素。这里我们看到内置仿函数变得异常复杂，而且程序员不得不接受使用非标准库函数的事实（compose2）。这对于需要移植性的程序来说，是非常难以让人接受的。即使之前曾经很多人对内置仿函数这样的做法点头称赞，

但现实情况下可能人人都必须承认：lambda 版本的实现非常的清晰，而且这一次代码量甚至少于内置仿函数的版本，简直无可挑剔。当然，这还不是全部，我们来看下一个例子，如代码清单 7-33 所示。

代码清单 7-33

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

vector<int> nums;

void Add(const int val){
    auto print = [&]{
        for (auto s: nums){ cout << s << '\t'; }
        cout<< endl;
    };

    // 传统的 for 循环方式
    for (auto i = nums.begin(); i != nums.end(); ++i){
        *i = *i + val;
    }
    print();

    // 试一试 for_each 及内置仿函数
    for_each(nums.begin(), nums.end(), bind2nd(plus<int>(), val));
    print();

    // 实际这里需要使用 STL 的一个变动性算法：transform
    transform(nums.begin(), nums.end(), nums.begin(), bind2nd(plus<int>(), val));
    print();

    // 不过在 lambda 的支持下，我们还是可以只使用 for_each
    for_each(nums.begin(), nums.end(), [=](int &i){
                i += val;
            });
    print();
}


int main(){
    for (int i = 0; i < 10; i++){
        nums.push_back(i);
    }
    Add(10);
    return 1;
}
```

```
编译选项：g++ 7-3-17.cpp -std=c++11
```

在代码清单 7-33 所示的例子中，我们试图改变 vector 中的内容，即将 vector 中所有的元素的数值加 10。这里我们还是使用了传统的 for 方式、内置仿函数的方式，以及 lambda 的方式。此外，为了方便调试，我们使用了一个 lambda 函数来打印局部运行的结果。

在机器上编译运行代码清单 7-33，可以看到如下运行结果：

```
10      11      12      13      14      15      16      17      18      19
10      11      12      13      14      15      16      17      18      19
20      21      22      23      24      25      26      27      28      29
30      31      32      33      34      35      36      37      38      39
```

这里我们注意到，结果的第二行和第一行没有区别。仔细查过资料之后，我们发现，内置的仿函数 plus<int>() 仅仅将加法结果返回，为了将返回结果再应用于 vector nums，通常情况下，我们需要使用 transform 这个算法。如我们第三段代码所示，transform 会遍历 nums，并将结果写入 nums.begin() 出首地址的目标区（第三参数）。

事实上，在书写 STL 的时候人们总是会告诫新手 for_each 和 transform 之间的区别，因为 for_each 并不像 transform 一样写回结果。这在配合 STL 内置的仿函数的时候就会有些使用上的区别。但在 C++11 的 lambda 来临的时候，这样的困惑就变少了。因为内置的仿函数并非必不可少，lambda 中包含的代码逻辑一目了然，使用 STL 算法的规则也因此变得简单多了。

我们来看最后一个 STL 的例子，如代码清单 7-34 所示。

代码清单 7-34

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

void Stat(vector<int> &v){
    int errors;
    int score;
    auto print = [&]{
        cout << "Errors: " << errors << endl
             << "Score: " << score << endl;
    };
    // 使用 accumulate 算法，需要两次循环
    errors = accumulate(v.begin(), v.end(), 0);
    score = accumulate(v.begin(), v.end(), 100, minus<int>());
    print();

    errors = 0;
    score = 100;
```

```
    // 使用 lambda 则只需要一次循环
    for_each(v.begin(), v.end(), [&](int i){
            errors += i;
            score -= i;
            });
    print();
}

int main(){
    vector<int> v(10);
    generate(v.begin(), v.end(), []{
            return rand() % 10;
            });
    Stat(v);
}
```

编译选项：g++ 7-3-18.cpp -std=c++11

代码清单 7-34 是一个区间统计的例子，通常情况下，我们可以使用 STL 的 accumulate 算法及内置仿函数来完成这个运算。从代码的简洁性上来看，这样做好像也不错。不过实际上我们产生了两次循环，一次计算 errors，一次计算 score。而使用 lambda 和万能的 for_each 的版本的时候，我们可以从源代码层将循环合并起来。事实上，我们可能在实际工作中能够合并的循环更多。如果采用 accumulate 的方式，而编译器不能合理有效地合并循环的话，我们可能就会遭受一定的性能损失（比如 cache 方面）。

可以看到，对于 C++ 的 STL 和泛型编程来讲，lambda 存在的意义重大，尤其是对于 STL 的算法方面，lambda 的出现使得学习使用 STL 算法的代价大大降低，甚至会改变一些使用 STL 算法的思维。当然，在一些情况下，lambda 还能在代码自说明、性能上具有一定的优势。这些都是很多 C++ 程序员，尤其是喜欢泛型编程的 C++ 程序员梦寐以求的。

7.3.7 更多的一些关于 lambda 的讨论

lambda 被纳入 C++ 语言之后，很多人认为 lambda 让 C++11 语言看起来更加复杂。一来 lambda 的语法与之前的 C++ 语法相比起来显得有些独特，二来其基于仿函数的实现，让初学者会感觉一时间找不到它的用途。不过，在考虑过编写仿函数或者使用 STL 内置的仿函数的复杂性之后，大多数人会肯定 lambda 的应用价值。

不过要完全用好 lambda，必须了解一些 lambda 的特质。比如 lambda 和仿函数之间的取舍，如何有效地使用 lambda 的捕捉列表等。

首先必须了解的是，在现有 C++11 中，lambda 并不是仿函数的完全替代者。这一点很大程度上是由 lambda 的捕捉列表的限制造成的。在现行 C++11 标准中，捕捉列表仅能捕捉父作用域的自动变量，而对超出这个范围的变量，是不能被捕捉的。我们可以看看代码清单

7-35 所示的例子。

代码清单 7-35

```
int d = 0;
int TryCapture() {
    auto ill_lambda = [d]{};
}

// 编译选项 :g++ -std=c++11 -c 7-3-19.cpp
```

代码清单 7-35 所示的例子在一些编译器（如 g++）上可以编译通过，但程序员会得到一些编译器的警告，而一些严格遵守 C++11 语言规则的编译器则会直接报错。而如果我们采用仿函数，则不会有这样的限制，如代码清单 7-36 所示。

代码清单 7-36

```
int d = 0;

class healthyFunctor{
public:
    healthyFunctor(int d): data(d){}

    void operator () () const {}

private:
    int data;
};

int TryCapture() {
    healthyFunctor hf(d);
}

// 编译选项 :g++ -std=c++11 -c 7-3-20.cpp
```

在代码清单 7-36 中的 healthyFunctor 说明了两者的不同。更一般地讲，仿函数可以被定义以后在不同的作用域范围内取得初始值。这使得仿函数天生具有了跨作用域共享的特征。

总地来说，lambda 函数被设计的目的，就是要就地书写，就地使用。使用 lambda 的用户，更倾向于在一个屏幕里看到所有的代码，而不是依靠代码浏览工具在文件间找到函数的实现。而在封装的思维层面上，lambda 只是一种局部的封装，以及局部的共享。而需要全局共享的代码逻辑，我们则还是需要用函数（无状态）或者仿函数（有状态）封装起来。

简单地总结一下，使用 lambda 代替仿函数的应该满足如下一些条件：

- 是局限于一个局部作用域中使用的代码逻辑。

❑这些代码逻辑需要被作为参数传递。

此外，关于捕捉列表的使用也存在有很多的讨论。由于 [=], [&] 这些写法实在是太过方便了，有的时候，我们不会仔细思考其带来的影响就开始滥用，这也会造成一些意想不到的问题。

首先，我们来看一下 [=]，除去我们之前提过的，所有捕捉的变量在 lambda 声明一开始就被拷贝，且拷贝的值不可被更改，这两点需要程序员注意之外，还有一点就是拷贝本身。这点跟函数参数按值方式传递是一样的，如果不想带来过大的传递开销的话，可以采用引用传递的方式传递参数。

其次，我们再来看一下 [&]。如我们之前提到过的，通过引用方式传递的对象也会输出到父作用域中。同样的，父作用域对这些对象的操作也会传递到 lambda 函数中。因此，如果我们代码存在异步操作，或者其他可能改变对象的任何操作，我们必须确定其在父作用域及 lambda 函数间的关系，否则也会产生一些错误。

通常情况下，在使用 [=]、[&] 这些默认捕捉列表的时候，我们需要考察其性能、与父作用域如何关联等。捕捉列表是 lambda 最神奇也是最容易犯错的地方，程序员不能一味图方便了事。

7.4 本章小结

本章重点讲解了3个C++11中会给用户带来思维方式上转变的的特性：nullptr、=default/=deleted 以及 lambda 函数。

nullptr 这个新特性主要是用于指针初始化，C++11 标准通过引入新关键字 nullptr 排除字面常量 0 的二义性，使之成为指针初始化的标准方式。nullptr 可以安全地转换成各种指针，这使得 nullptr 使用简便，而 nullptr 不能转换为除指针外的任何类型的特性，又使得使用 nullptr 具有高度的安全性。程序员只需要简单将以前使用 NULL 的地方换成 nullptr 就可以在 C++11 编译器的支持下获得更佳的、更健壮的指针初始化代码。

而 default/deleted 函数则试图在 C++ 缺省函数是否生成上给程序员更多的控制力。通过显式缺省函数，我们可以保证自定义类型符合 POD 的定义，而通过显式删除缺省函数，我们可以禁止 class 的使用者使用不应使用的函数。不过除去 C++ 默认提供的缺省函数外，C++11 标准给予了显式缺省和显式删除更多的能力，比如生成一些并非默认提供的函数的缺省定义，或者禁止一些不必要的隐式类型转换。程序员甚至可以通过删除一些函数来完成编译时期的内存分配的检查。在这样的新特性的支持下，程序员尤其是类的编写者可以更加轻松地保证写出来的类具备所需的特性。

相比于之上两个新特性，lambda 对于 C++ 程序的编写具有更大的冲击力。

早在 C++98 时代，程序员会喜欢使用简单的 STL 部件，如容器等。而使用 STL 的算法（algorithm）的用户相对较少。这一方面固然是由于学习难度较大、简单的算法都可以手工实现造成的，另外一方面的原因则是由于 STL 的算法大多数是依赖于迭代器、仿函数这些较为高级概念造成的。尤其对于仿函数，用户自定义的仿函数需要遵循规则才能被重用。而使用 STL 自带的各种各样的仿函数使用范围和方式过于受限，也使得 STL 良好设计算法的由于“不接地气”而推广受限。

lambda 被设计出来的主要目的之一是简化仿函数的使用，虽然 lambda 的语法看起来不像是“典型的 C++”的，但一旦熟悉了之后，程序员就能准确地完成一个简单的、就地的、带状态的函数定义。虽然 lambda 可以跟仿函数的概念一一对应也实际由仿函数来实现，但理解 lambda 显然比仿函数更加容易。即使没有学习过仿函数的程序员也可以轻松接受 lambda。这样一来，lambda 的出现必然导致 STL 设计、使用方法上的改变，但其最终意义是让更多的程序员无困难地使用 STL 的各种调优后的算法，真正享受 STL 的全部力量。

此外，lambda 作为局部函数也会使得复杂代码的开发加速。通过 lambda 函数，程序员可以轻松地在函数内重用代码，而无需费心设计接口。事实上，曾经出现过一般重构的风潮，在那段时期，C++ 程序员被建议为任何重用的代码创建函数，而 lambda 局部函数的到来则带来了理性思考和折中实现的可能。当然，lambda 函数的出现也导致了函数的作用域在 C++11 中再次被细分，从而也使得 C++ 编程具备了更多的可能。

以上种种或许是 C++ 早就应该做到的，但一直未能实现，现在终于在 C++11 中，lambda 让我们看到了新的光芒。

第⑧章

融入实际应用

对于 C++ 这样的语言来说，除了具有普适、通用等特性外，在一些实际应用方面也是不容忽视的，比如最为典型的，也是非英语国家程序员体会最深的，与字符编码相关的种种问题。在 C++11 中，我们看到语言标准终于也对 Unicode 做了较多深入的支持，以切实地为不同语言服务。此外，以前在 C++ 编程中的各种讨人喜欢的编译器扩展，也开始逐渐被 C++11 收编或者规范化。而有了标准的支持，这些特性也将更加好用。本章中，读者可以了解 C++11 在这些方面做出的务实而有效的改进。

8.1 对齐支持

☞ 类别：部分人

8.1.1 数据对齐

在了解为什么数据需要对齐之前，我们可以回顾一下打印结构体的大小这个 C/C++ 中的经典案例。首先来看看代码清单 8-1 所示的这个例子。

代码清单 8-1

```
#include <iostream>
using namespace std;

struct HowManyBytes{
    char    a;
    int     b;
};

int main() {
    cout << "sizeof(char): " << sizeof(char) << endl;
    cout << "sizeof(int): " << sizeof(int) << endl;
    cout << "sizeof(HowManyBytes): " << sizeof(HowManyBytes) << endl;

    cout << endl;
    cout << "offset of char a: " << offsetof(HowManyBytes, a) << endl;
    cout << "offset of int b: " << offsetof(HowManyBytes, b) << endl;
    return 0;
```

```
}

// 编译选项 :g++ -std=c++11 8-1-1.cpp
```

在代码清单 8-1 中，结构体 HowManyBytes 由一个 char 类型成员 a 及一个 int 类型成员 b 组成。编译运行代码清单 8-1 所示的例子，我们在实验机平台上会得到如下结果：

```
sizeof(char): 1
sizeof(int): 4
sizeof(HowManyBytes): 8

offset of char a: 0
offset of int b: 4
```

可以看到，在我们的实验机上，a 和 b 两个数据的长度分别为 1 字节和 4 字节，不过当我们使用 sizeof 来计算 HowManyBytes 这个结构体所占用的内存空间时，看到其值为 8 字节。其中似乎多出来了 3 字节没有使用的空间。

出现这个现象主要是由于数据对齐要求导致的。通常情况下，C/C++ 结构体中的数据会有一定的对齐要求。在这个例子中，可以通过 offsetof 查看成员的偏移的方式来检验数据的对齐方式。这里，成员 b 的偏移是 4 字节，而成员 a 只占用了 1 字节内存空间，这意味着 b 并非紧邻着 a 排列。事实上，在我们的平台定义上，C/C++ 的 int 类型数据要求对齐到 4 字节，即要求 int 类型数据必须放在一个能够整除 4 的地址上；而 char 要求对齐到 1 字节。这就造成了成员 a 之后的 3 字节空间被空出，通常我们也称因为对齐而造成的内存留空为填充数据（padding data）。

在 C++ 中，每个类型的数据除去长度等属性之外，都还有一项“被隐藏”属性，那就是对齐方式。对于每个内置或者自定义类型，都存在一个特定的对齐方式。对齐方式通常是一个整数，它表示的是一个类型的对象存放的内存地址应满足的条件。在这里，我们简单地将其称为对齐值。

对齐的数据在读写上会有性能上的优势。比如频繁使用的数据如果与处理器的高速缓存器大小对齐，有可能提高缓存性能。而数据不对齐可能造成一些不良的后果，比较严重的当属导致应用程序退出。典型的，如在有的平台上，硬件将无法读取不按字对齐的某些类型数据，这个时候硬件会抛出异常（如 bus error）来终止程序。而更为普遍的，在一些平台上，不按照字对齐的数据会造成数据读取效率低下。因此，在程序设计时，保证数据对齐是保证正确有效读写数据的一个基本条件。

虽然从语言设计者的角度而言，将对齐方式掩盖起来会使得语言更具有亲和力。但通常由于底层硬件的设计或用途不同，以及编程语言本身在基本（内置）类型的定义上的不同，相同的类型定义在不同的平台上会有不同的长度，以及不同的对齐要求。虽然系统设计者常常会在应用程序二进制接口中（Application Binary Interface，ABI）详细规定在特定平台上数

据长度、数据对齐方式的相关信息，但是这两者存在着平台差异性则是不争的事实。在 C++ 语言中，我们可以通过 sizeof 查询数据长度，但 C++ 语言却没有对对齐方式有关的查询或者设定进行标准化，而语言本身又允许自定义类型、模板等诸多特性。编译器无法完全找到正确的对齐方式，这会在使用时造成困难。让我们来看一下代码清单 8-2 所示的这个例子。

代码清单 8-2

```
#include <iostream>
using namespace std;

// 自定义的 ColorVector，拥有 32 字节的数据
struct ColorVector {
    double r;
    double g;
    double b;
    double a;
};

int main() {
    // 使用 C++11 中的 alignof 来查询 ColorVector 的对齐方式
    cout << "alignof(ColorVector): " << alignof(ColorVector) << endl;
    return 1;
}

// 编译选项 :clang++ -std=c++11 8-1-2.cpp
```

在代码清单 8-2 所示的例子中，我们使用了 C++11 标准定义的 alignof 函数来查看数据的对齐方式。编译运行代码清单 8-2，我们可以看到 ColorVector 在实验机上依然是对齐到 8 字节的地址边界上。

```
alignof(ColorVector): 8
```

这与我们设计 ColorVector 的原意是不同的。现在的计算机通常会支持许多向量指令，而 ColorVector 正好是 4 组 8 字节的浮点数数据，很有潜力改造为能直接操作的向量数据。这样一来，为了能够高效地读写 ColorVector 大小的数据，我们最好能将其对齐在 32 字节的地址边界上。

在代码清单 8-3 所示的例子中，我们利用 C++11 新提供的修饰符 alignas 来重新设定 ColorVector 的对齐方式。

代码清单 8-3

```
#include <iostream>
using namespace std;

// 自定义的 ColorVector，对齐到 32 字节的边界
struct alignas(32) ColorVector {
```

```
    double r;
    double g;
    double b;
    double a;
};

int main() {
    // 使用 C++11 中的 alignof 来查询 ColorVector 的对齐方式
    cout << "alignof(ColorVector): " << alignof(ColorVector) << endl;
    return 1;
}

// 编译选项 :g++ -std=c++11 8-1-3.cpp
```

编译运行代码清单 8-3 所示的代码，我们会得到如下结果：

```
alignof(ColorVector): 32
```

正如我们在代码清单 8-3 中所看到的，指定数据 ColorVector 对齐到 32 字节的地址边界上，只需要声明 alignas(32) 即可。接下来我们会详细讨论 C++11 对对齐的支持。

8.1.2 C++11 的 alignof 和 alignas

如同我们在上一节中看到的，C++11 在新标准中为了支持对齐，主要引入两个关键字：操作符 alignof、对齐描述符（alignment-specifier）alignas。

操作符 alignof 的操作数表示一个定义完整的自定义类型或者内置类型或者变量，返回的值是一个 std::size_t 类型的整型常量。如同 sizeof 操作符一样，alignof 获得的也是一个与平台相关的值。我们可以看看代码清单 8-4 所示的例子。

代码清单 8-4

```
#include <iostream>
using namespace std;

class InComplete;
struct Completed{};

int main(){
    int a;
    long long b;
    auto & c = b;
    char d[1024];

    // 对内置类型和完整类型使用 alignof
    cout << alignof(int) << endl          //  4
        << alignof(Completed) << endl;    //  1
```

```
    // 对变量、引用或者数组使用 alignof
    cout << alignof(a) << endl          //  4
         << alignof(b) << endl          //  8
         << alignof(c) << endl          //  8，与 b 相同
         << alignof(d) << endl;         //  1，与元素要求相同

    // 本句无法通过编译，Incomplete 类型不完整
    // cout << alignof(Incomplete) << endl;
}

// 编译选项 :g++ -std=c++11 8-1-4.cpp
```

使用 alignof 很简单，基本上没有什么特别的限制。不过在代码清单 8-4 中，类型定义不完整的 class InComplete 是无法通过编译的。其他的规则则基本跟大多数人想象的相同：引用 c 与其引用的数据 b 对齐值相同，数组的对齐值由其元素决定。

我们再来看看对齐描述符 alignas。事实上，alignas 既可以接受常量表达式，也可以接受类型作为参数，比如

```
alignas(double) char c;
```

也是合法的描述符。其使用效果跟

```
alignas(alignof(double)) char c;
```

是一样的。

注意 在 C++11 标准之前，我们也可以使用一些编译器的扩展来描述对齐方式，比如 GNU 格式的 __attribute__((__aligned__(8))) 就是一个广泛被接受的版本。

我们在使用常量表达式作为 alignas 的操作符的时候，其结果必须是以 2 的自然数幂次作为对齐值。对齐值越大，我们称其对齐要求越高；而对齐值越小，其对齐要求也越低。由于 2 的幂次的关系，能够满足严格对齐要求的对齐方式也总是能够满足要求低的对齐值的。

在 C++11 标准中规定了一个“基本对齐值”（fundamental alignment）。一般情况下其值通常等于平台上支持的最大标量类型数据的对齐值（常常是 long double）。我们可以通过 alignof(std::max_align_t) 来查询其值。而像我们在代码清单 8-3 中设定 ColorVector 对齐值到 32 字节（超过标准对齐）的做法称为扩展对齐（extended alignment）。不过即使使用了扩展对齐，也并非意味着程序员可以随心所欲。对于每个平台，系统能够支持的对齐值总是有限的，程序中如果声明了超过平台要求的对齐值，则按照 C++ 标准该程序是不规范的（ill-formed），这可能会导致未知的编译时或者运行时错误。因此程序员应该定义合理的对齐值，否则可能会遇到一些麻烦。

对齐描述符可以作用于各种数据。具体来说，可以修饰变量、类的数据成员等，而位域（bit field）以及用 register 声明的变量则不可以。我们可以看看 C++11 标准中的这个例子，如

代码清单 8-5 所示。

代码清单 8-5

```
alignas(double) void f(); // 错误：alignas 不能修饰函数
alignas(double) unsigned char c[sizeof(double)]; // 正确
extern unsigned char c[sizeof(double)];
alignas(float)
    extern unsigned char c[sizeof(double)]; // 错误：不同对齐方式的变量定义

// 编译选项 :clang++ 8-1-5.cpp -c -std=c++11
```

对于代码清单 8-5 所示的例子，标准给出了建议的答案（如注释所示）。C++11 标准建议用户在声明同一个变量的时候使用同样的对齐方式以免发生意外。不过 C++11 并没有规定声明变量采用了不同的对齐方式就终止编译器的编译。在编写本书时，clang++ 编译器对该例就没有终止编译，而是使用了最严格的对齐方式作为 c 的最终对齐方式。读者可以试试自己的编译环境，看一下编译器是如何处理的。

我们再来看一个例子，这个例子中我们采用了模板的方式来实现一个固定容量但是大小随着所用的数据类型变化的容器类型，如代码清单 8-6 所示。

代码清单 8-6

```
#include <iostream>
using namespace std;

struct alignas(alignof(double)*4) ColorVector {
    double r;
    double g;
    double b;
    double a;
};

// 固定容量的模板数组
template <typename T>
class FixedCapacityArray {
public:
    void push_back(T t) { /* 在 data 中加入 t 变量 */}
    // ...
    // 一些其他成员函数、成员变量等
    // ...
    char alignas(T) data[1024] = {0};
    //int length = 1024 / sizeof(T);
};

int main() {
    FixedCapacityArray<char> arrCh;
    cout << "alignof(char): " << alignof(char) << endl;
```

```
    cout << "alignof(arrCh.data): " << alignof(arrCh.data) << endl;

    FixedCapacityArray<ColorVector> arrCV;
    cout << "alignof(ColorVector): " << alignof(ColorVector) << endl;
    cout << "alignof(arrCV.data): " << alignof(arrCV.data) << endl;
    return 1;
}

// 编译选项 :clang++ 8-1-6.cpp -std=c++11
```

代码清单 8-6 修改自代码清单 8-3，在本例中，FixedCapacityArray 固定使用 1024 字节的空间，但由于模板的存在，可以实例化为各种版本。这样一来，我们可以在相同的内存使用量的前提下，做出多种类型（内置或者自定义）版本的数组。

如我们之前提到的一样，为了有效地访问数据，必须使得数据按照其固有特性进行对齐。对于 arrCh，由于数组中的元素都是 char 类型，所以对齐到 1 就行了，而对于我们定义的 arrCV，必须使其符合 ColorVector 的扩展对齐，即对齐到 8 字节的内存边界上。在这个例子中，起到关键作用的代码是下面这一句：

```
char alignas(T) data[1024] = {0};
```

该句指示 data[1024] 这个 char 类型数组必须按照模板参数 T 的对齐方式进行对齐。

编译运行该例子后，可以在实验机上得到如下结果：

```
alignof(char): 1
alignof(arrCh.data): 1
alignof(ColorVector): 32
alignof(arrCV.data): 32
```

如果我们去掉 alignas(T) 这个修饰符，代码清单 8-6 的运行结果会完全不同，具体如下：

```
alignof(char): 1
alignof(arrCh.data): 1
alignof(ColorVector): 32
alignof(arrCV.data): 1
```

可以看到，由于 char 数组默认对齐值为 1，会导致 data[1024] 数组也对齐到 1。这肯定不是编写 FixedCapacityArray 的程序员愿意见到的。

事实上，在 C++11 标准引入 alignas 修饰符之前，这样的固定容量的泛型数组有时可能遇到因为对齐不佳而导致的性能损失（甚至程序错误），这给库的编写者带来了很大的困扰。而引入 alignas 能够解决这些移植性的困难，这可能也是 C++ 标准委员会决定不再“隐藏”变量的对齐方式的原因之一。

C++11 对于对齐的支持并不限于 alignof 操作符及 alignas 描述符。在 STL 库中，还内建了 std::align 函数来动态地根据指定的对齐方式调整数据块的位置。该函数的原型如下：

```
void* align( std::size_t alignment, std::size_t size, void*& ptr, std::size_t& space );
```

该函数在 ptr 指向的大小为 space 的内存中进行对齐方式的调整，将 ptr 开始的 size 大小的数据调整为按 alignment 对齐。我们可以看看代码清单 8-7 所示的这个例子。

代码清单 8-7

```
#include <iostream>
#include <memory>
using namespace std;

struct ColorVector {
    double r;
    double g;
    double b;
    double a;
};

int main() {
    size_t const size = 100;
    ColorVector * const vec = new ColorVector[size];

    void* p = vec;
    size_t sz = size;

    void* aligned = align(alignof(double) * 4, size, p, sz);
    if (aligned != nullptr)
        cout << alignof(p) << endl;
}
```

代码清单 8-7 尝试将 vec 中的内容按 alignof(double)*4 的对齐值进行对齐（不过在编写本书的时候，我们的编译器还没有支持 std::align 这个新特性，因此代码清单 8-7 仅供参考）。

事实上，C++11 还在标准库中提供了 aligned_storage 及 aligned_union 供程序员使用。两者的原型如下：

```
template< std::size_t Len, std::size_t Align = /*default-alignment*/ >
struct aligned_storage;
template< std::size_t Len, class... Types >
struct aligned_union;
```

aligned_storage 的第一个参数规定了 aligned_storage 的大小，第二个参数则是其对齐值。我们可以通过代码清单 8-8 所示的这个例子说明它们的用途。

代码清单 8-8

```
#include <iostream>
#include <type_traits>
using namespace std;
```

```
    // 一个对齐值为 4 的对象
    struct IntAligned{
      int a;
      char b;
    };

    // 使用 aligned_storage 使其对齐要求更加严格
    typedef  aligned_storage<sizeof(IntAligned),alignof(long  double)>::type
StrictAligned;

    int main() {
      StrictAligned sa;
      IntAligned* pia = new (&sa) IntAligned;
      cout << alignof(IntAligned) << endl;       // 4
      cout << alignof(StrictAligned) << endl;    // 16
      cout << alignof(*pia) << endl;             // 4
      cout << alignof(sa) << endl;               // 16
      return 0;
    }

    // 编译选项 :g++ -std=c++11 8-1-8.cpp
```

在代码清单 8-8 中，我们使用了一个 placement new 来使得 StrictAligned 存储了本来应该只需要按照 4 字节对齐的 IntAligned 对象。虽然 StrictAligned 对象 sa 的内容与 IntAligned 类型指针 pia 所指向的对象完全相同，但通过这样的声明，却产生了比 *pia 更加严格的类型对齐要求（本例中为 16）。因此虽然最后 IntAligned 对象的对齐方式没有发生改变，但实际上却被更严格地对齐了。

有的时候，一个类型声明的代码较长，可能需要程序员上下翻页来阅读（虽然面向对象的规则并不推荐这样做，但在大型项目中，代码很长的类型声明并不少见），通常为了对齐，程序员不得不自己写一些填充来保证其大小。除了代码较难阅读外，每个系统上结构体、联合体的对齐规则也可能不一样，这在代码维护上是一种挑战。如果后加入类型成员的程序员没有注意到这里的对齐要求或者代码编写不慎，很可能会添加了导致对齐改变的代码（对齐变严格一般不是问题，但反之则可能是问题）。

改进的方法可以是使用 alignas 描述符，假如代码清单 8-8 中的 IntAligned 是一个代码很长的 struct 声明，那么对其使用一个 alignas 描述符就是一个可行的方法。不过很多时候，数据的声明是需要共享的，假如超长的 IntAligned 需要在支持和不支持 alignas 的编译环境下共享（典型的，要在老的 C 环境及 C++11 环境下共享头文件），那么使用 aligned_storage 则是一个可行的方法，因为 aligned_storage 可以在产生对象的实例时对对齐方式做出一定的保证。这无疑对“有历史”的代码的重用、维护很有意义。

aligned_union 的用法也基本与此相同。只不过 aligned_union 使用了变长模板参数，程序

员可以根据需要填入多种类型，最后 aligned_union 对象的对齐要求会是多个类型中要求最为严格的一个。

可以看到，在新的 C++11 标准中，对对齐方式的支持是全方面的，无论是查看（alignof）、设定（alignas），还是 STL 库函数（std::align）或是 STL 库模板类型（aligned_storage, aligned_union），程序员都可以找到对应的方法。这使得一些非标准的设定对齐方式的做法规范统一，真正满足程序员在可移植性上的要求。事实上，程序的可移植性还有很多的相关问题，接下来要讲到的通用属性，就与对齐方式有很多关联。

8.2 通用属性

☞ **类别：部分人**

8.2.1 语言扩展到通用属性

随着 C++ 语言的演化和编译器的发展，人们常会发现标准提供的语言能力不能完全满足要求。于是编译器厂商或组织为了满足编译器客户的需求，设计出了一系列的语言扩展（language extension）来扩展语法。这些扩展语法并不存在于 C++/C 标准中，却有可能拥有较多的用户。有的时候，新的标准也会将广泛使用的语言扩展纳入其中。

扩展语法中比较常见的就是“属性”（attribute）。属性是对语言中的实体对象（比如函数、变量、类型等）附加一些的额外注解信息，其用来实现一些语言及非语言层面的功能，或是实现优化代码等的一种手段。

不同编译器有不同的属性语法。比如对于 g++，属性是通过 GNU 的关键字 __attribute__ 来声明的。程序员只需要简单地声明：

```
__attribute__ ((attribute-list))
```

即可为程序中的函数、变量和类型设定一些额外信息，以便编译器可以进行错误检查和性能优化等。我们可以看看代码清单 8-9 所示的例子。

代码清单 8-9

```
extern int area(int n) __attribute__((const));

int main() {
    int i;
    int areas = 0;
    for (i = 0; i < 10; i++) {
        areas += area(3) * i;
    }
}
```

```
// 编译选项 :g++ -c 8-2-1.cpp
```

这里的 const 属性告诉编译器：本函数返回值只依赖于输入，不会改变任何函数外的数据，因此没有任何副作用。在了解该信息的情况下，编译器可以对 area 函数进行优化处理。area(3) 的值只需要计算一次，编译之后可以将 area(3) 视为循环中的常量而只使用其计算结果，从而大大提高了程序的执行性能。

注意 事实上，在 GNU 对 C/C++ 的扩展中我们可以看到很多不同的 __attribute__ 属性。常见的如 format、noreturn、const 和 aligned 等，具体含义和用法读者可以参考 GNU 的在线文档 http://gcc.gnu.org/onlinedocs/。

而在 Windows 平台上，我们会找到另外一种关键字 __declspec。__declspec 是微软用于指定存储类型的扩展属性关键字。用户只要简单地在声明变量时加上：

```
__declspec ( extended-decl-modifier )
```

即可设定额外的功能。以对齐方式为例，在 C++11 之前，微软平台的程序员可以使用 __declspec(align(x)) 来控制变量的对齐方式，如代码清单 8-10 所示。

代码清单 8-10

```
__declspec(align(32)) struct Struct32 {
    int i;
    double d;
};
```

在代码清单 8-10 中，结构体 Struct32 被对齐到 32 字节的地址边界，其起始地址必须是 32 的倍数。这跟 C++11 中 alignas 的效果是一样的。

注意 同样的，微软也定义了很多 __declspec 属性，如 noreturn、oninline、align、dllimport、dllexport 等，具体含义和用法可以参考微软网站上的介绍：http://msdn.microsoft.com/en-US/library/。

事实上，在扩展语言能力的时候，关键字往往会成为一种选择。GNU 和微软只能选择“属性”这样的方式，是为了尽可能避免与用户自定义的名称冲突。同样，在 C++11 标准的设立过程中，也面临着关键字过多的问题。于是 C++11 语言制定者决定增加了通用属性这个特性。不过 C++11 的通用属性设计跟 GNU 和微软都不一样，至少直观地看来，其更加简洁。

8.2.2 C++11 的通用属性

C++11 语言中的通用属性使用了左右双中括号的形式：

```
[[ attribute-list ]]
```

这样设计的好处是：既不会消除语言添加或者重载关键字的能力，又不会占用用户空间的关键字的名字空间。

语法上，C++11 的通用属性可以作用于类型、变量、名称、代码块等。对于作用于声明的通用属性，既可以写在声明的起始处，也可以写在声明的标识符之后。而对于作用于整个语句的通用属性，则应该写在语句起始处。而出现在以上两种规则描述的位置之外的通用属性，作用于哪个实体跟编译器具体的实现有关。

我们可以看几个例子。第一个是关于通用属性应用于函数的，具体如下：

```
[[ attr1 ]] void func [[ attr2 ]] ();
```

这里，[[attr1]] 出现在函数定义之前，而 [[attr2]] 则位于函数名称之后，根据定义，[[attr1]] 和 [[attr2]] 均可以作用于函数 [func]。下一个是数组的例子：

```
[[ attr1 ]] int array [[ attr2 ]] [10];
```

这跟第一个例子很类似，根据定义，[[attr1]] 和 [[attr2]] 均可以作用于数组 array。下面这个例子则稍显复杂：

```
[[ attr1 ]] class C [[ attr2 ]] { } [[ attr3 ]] c1 [[ attr4 ]], c2 [[ attr5 ]];
```

这个例子声明了类 C 及其类型的变量 c1 和 c2。本语句中，一共有 5 个不同的属性。按照 C++11 的定义，[[attr1]] 和 [[attr4]] 会作用于 c1，[[attr1]] 和 [[attr5]] 会作用于 c2，[[attr2]] 出现在声明之后，仅作用于类 C，而 [[attr3]] 所作用的对象则跟具体实现有关。

下面是一个 switch-case 加标签的例子：

```
[[ attr1 ]] L1:

switch(value){
    [[ attr2 ]] case 1:  // do something...
    [[ attr3 ]] case 2:  // do something...
    [[ attr4 ]] break;
    [[ attr5 ]] default: // do something...
}

[[ attr6 ]] goto L1;
```

这里，[[attr1]] 作用于标签 L1，[[attr2]] 和 [[attr3]] 作用于 case 1 和 case 2 表达式，[[attr4]] 作用于 break，[[attr5]] 作用于 default 表达式，[[attr6]] 作用于 goto 语句。下面的 for 语句也是类似的：

```
[[ attr1 ]] for( int i = 0; i < top; i++ ) {
   // do something...
}
[[ attr2 ]] return top;
```

这里，[[attr1]] 作用于 for 表达式，[[attr2]] 作用于 return。下面是函数有参数的情况：

```
[[ attr1 ]] int func([[ attr2 ]] int i, [[ attr3 ]] int j)
{
    // do something
    [[ attr4 ]] return i + j;
}
```

[[attr1]] 作用于函数 func，[[attr2]] 和 [[attr3]] 分别作用于整型参数 i 和 j，[[attr4]] 作用于 return 语句。

事实上，在现有 C++11 标准中，只预定义了两个通用属性，分别是 [[noreturn]] 和 [[carries_dependency]]。而在 C++11 标准委员会的最初提案中，还包含了形如 [[final]]、[[override]]、[[restrict]]、[[hides]]、[[base_check]] 等通用属性。不过最终，标准委员会只通过了以上两个，原因大概有以下几点：

- final、override、restrict 等是 C++ 语言中需要支持的语言特性。通用属性从设计上讲，是可忽略的属性，其设计的目的主要是为了帮助编译器更好地检查代码中的错误或帮助编译器更好地优化代码。因此，语义相关的部分还是需要使用在关键字上。
- 预定义的通用属性应该是可移植的。一旦预定义了过多的通用属性，会导致 C++ 代码的可移植性变弱。
- C 语言是没有通用属性的。

虽然看起来通用属性的使用受到了一些限制，但至少其语法规则为编译器厂商或组织提供了实现不同属性的办法。

8.2.3 预定义的通用属性

如上文所述，C++11 预定义的通用属性包括 [[noreturn]] 和 [[carries_dependency]] 两种。

[[noreturn]] 是用于标识不会返回的函数的。这里必须注意，不会返回和没有返回值的（void）函数的区别。没有返回值的 void 函数在调用完成后，调用者会接着执行函数后的代码；而不会返回的函数在被调用完成后，后续代码不会再被执行。

[[noreturn]] 主要用于标识那些不会将控制流返回给原调用函数的函数，典型的例子有：有终止应用程序语句的函数、有无限循环语句的函数、有异常抛出的函数等。通过这个属性，开发人员可以告知编译器某些函数不会将控制流返回给调用函数，这能帮助编译器产生更好的警告信息，同时编译器也可以做更多的诸如死代码消除、免除为函数调用者保存一些特定寄存器等代码优化工作。

我们可以看看代码清单 8-11 所示的这个例子。

代码清单 8-11

```
void DoSomething1();
void DoSomething2();

[[ noreturn ]] void ThrowAway() {
    throw "expection"; // 控制流跳转到异常处理
}

void Func(){
    DoSomething1();
    ThrowAway();
    DoSomething2(); // 该函数不可到达
}

// 编译选项 :clang++ -std=c++11 -c 8-2-3.cpp
```

在代码清单 8-11 中，由于 ThrowAway 抛出了异常，DoSomething2 永远不会被执行。这个时候将 ThrowAway 标记为 noreturn 的话，编译器会不再为 ThrowAway 之后生成调用 DoSomething2 的代码。当然，编译器也可以选择为 Func 函数中的 DoSomething2 做出一些警告以提示程序员这里有不可到达的代码。

不返回的函数除了是有异常抛出的函数外，还有可能是有终止应用程序语句的函数，或是有无限循环语句的函数等。事实上，在 C++11 的标准库中，我们都能看到形如：

```
[[noreturn]] void abort(void) noexcept;
```

这样的函数声明。这里声明的是最常见的 abort 函数。abort 总是会导致程序运行的停止，甚至连自动变量的析构函数以及本该在 atexit() 时调用的函数全都不调用就直接退出了。因此声明为 [[noreturn]] 是有利于编译器优化的。

不过程序员还是应该小心使用 [[noreturn]]，也尽量不要对可能会有返回值的函数使用 [[noreturn]]。代码清单 8-12 所示的是一个错误使用 [[noreturn]] 的例子。

代码清单 8-12

```
#include <iostream>
using namespace std;

[[ noreturn ]] void Func(int i){
    // 当参数 i 的值为 0 时，该函数行为不可估计
    if (i < 0)
        throw "negative";
    else if (i > 0)
        throw "positive";
}

int main(){
```

```
    Func(0);
    cout << "Returned" << endl; // 无法执行该句
    return 1;
}

// 编译选项 :clang++ -std=c++11 8-2-4.cpp
```

代码清单 8-12 的例子中，Func 调用后的打印语句永远不会被执行，因为 Func 被声明为 [[noreturn]]。不过由于函数作者的疏忽，忘记了 i == 0 时的状况，因此在 i == 0 时，Func 运行结束时还是会返回 main 的。在我们的实验机上，编译运行该例子会在运行时发生“段错误”。当然，具体的错误情况可能会根据编译器和运行时环境的不同而有所不同。不过总的来说，程序员必须审慎使用 [[noreturn]]。

另外一个通用属性 [[carries_dependency]] 则跟并行情况下的编译器优化有关。事实上，[[carries_dependency]] 主要是为了解决弱内存模型平台上使用 memory_order_consume 内存顺序枚举问题。

如我们在第 6 章里讲到的，memory_order_consume 的主要作用是保证对当前原子类型数据的读取操作先于所有之后关于该原子变量的操作完成，但它不影响其他原子操作的顺序。要保证这样的“先于发生”的关系，编译器往往需要根据 memory_model 枚举值在原子操作间构建一系列的依赖关系，以减少在弱一致性模型的平台上产生内存栅栏。不过这样的关系则往往会由于函数的存在而被破坏。比如下面的代码：

```
atomic<int*> a;
...
int* p = (int *)a.load(memory_order_consume);
func(p);
```

上面的代码中，编译器在编译时可能并不知道 func 函数的具体实现，因此，如果要保证 a.load 先于任何关于 a（或是 p）的操作发生，编译器往往会在 func 函数之前加入一条内存栅栏。然而，如果 func 的实现是：

```
void func(int * p) {
   // ... 假设 p2 是一个 atomic<int*> 的变量
   p2.store(p, memory_order_release)
}
```

那么对于 func 函数来说，由于 p2.store 使用了 memory_order_release 的内存顺序，因此，p2.store 对 p 的使用会被保证在任何关于 p 的使用之后完成。这样一来，编译器在 func 函数之前加入的内存栅栏就变得毫无意义，且影响了性能。同样的情况也会发生在函数返回的时候。

而解决的方法正是使用 [[carries_dependency]]。该通用属性既可以标识函数参数，又可以标识函数的返回值。当标识函数的参数时，它表示数据依赖随着参数传递进入函数，即不需要产生内存栅栏。而当标识函数的返回值时，它表示数据依赖随着返回值传递出函数，同

样也不需要产生内存栅栏。更具体的我们可以看看代码清单 8-13 所示的例子。

代码清单 8-13

```
#include <iostream>
#include <atomic>
using namespace std;

atomic<int*> p1;
atomic<int*> p2;
atomic<int*> p3;
atomic<int*> p4;

void func_in1(int* val) {
    cout << *val << endl;
}

void func_in2(int* [[carries_dependency]] val) {
    p2.store(val, memory_order_release);
    cout << *p2 << endl;
}

[[carries_dependency]] int* func_out() {
    return (int *)p3.load(memory_order_consume);
}

void Thread() {
    int* p_ptr1 = (int *)p1.load(memory_order_consume);      // L1
    cout << *p_ptr1 << endl;// L2

    func_in1(p_ptr1);          // L3
    func_in2(p_ptr1);          // L4

    int * p_ptr2 = func_out();  // L5
    p4.store(p_ptr2, memory_order_release); // L6
    cout << *p_ptr2 << endl;
}

// 编译选项 :g++ -std=c++11 8-2-5.cpp -c
```

在代码清单 8-13 中，L1 句中，p1.load 采用了 memory_order_consume 的内存顺序，因此任何关于 p1 或者 p_ptr1 的原子操作，必须发生在 L1 句之后。这样一来，L2 将由编译器保证其执行必须在 L1 之后（通过编译器正确的指令排序和内存栅栏）。而当编译器在处理 L3 时，由于 func_in1 对于编译器而言并没有声明 [[carries_dependency]] 属性，编译器则可能采用保守的方法，在 func_in1 调用表达式之前插入内存栅栏。而编译器在处理 L4 句时，由于函数 func_in2 使用了 [[carries_dependency]]，编译器则会假设函数体内部会正确地处理内存顺序，因此不再产生内存栅栏指令。事实上 func_in2 中也由于 p2.store 使用了内存顺序

memory_order_release，因而不会产生任何的问题。而当编译器处理 L5 句时，由于 func_out 的返回值使用了 [[carries_dependency]]，编译器也不会在返回前为 p3.load(memory_order_consume) 插入内存栅栏指令去保证正确的内存顺序。而在 L6 行中，我们看到 p4.store 使用了 memory_order_release，因此 func_out 不产生内存栅栏也是毫无问题的。

事实上，本书编写时 [[carries_dependency]] 还没有被编译器支持，而对一些强内存模型的平台来说，编译器也常常会忽略该通用属性，因此其可用性比较有限。不过与 [[noreturn]] 相同的是，[[carries_dependency]] 只是帮助编译器进行优化，这符合通用属性设计的原则。当读者使用的平台是弱内存模型的时候，并且很关心并行程序的执行性能时，可以考虑使用 [[carries_dependency]]。

8.3 Unicode 支持

☞ 类别：所有人

8.3.1 字符集、编码和 Unicode

在了解 Unicode 之前，我们先回顾一下计算机表示信息的方式。无论是存储器中的晶体管通断，还是磁盘中磁畴的极性，或者是光盘中的坑槽，计算机总是使用两种不同的状态来作为基本信息，即二进制信息。而要标识现实生活中更为复杂的实体，则需要通过多个这样的基本信息的组合来完成。在计算机中，首当其冲需要被标识的就是字符。为了使二进制组合标识字符的方法在不同设计的计算机间通用，就迫切需要统一的字符编码方法。于是在 20 世纪 60 年代的时候，现在使用最为广泛的 ASCII 字符编码就出现了。

在 ANSI 颁布的标准中，基本 ASCII 的字符使用了 7 个二进制位进行标识，这意味着总共可以标识 128 种不同的字符。这对英文字符（以及一些控制字符、标点符号等）来说绰绰有余，不过随着计算机在全世界的普及，非字符构成的语言（如中文）也需要得到支持，128 个字符对于全世界众多语言而言就显得力不从心了。

到了 20 世纪 90 年代，ISO 与 Unicode 两个组织共同发布了能够唯一地表示各种语言中的字符的标准。通常情况下，我们将一个标准中能够表示的所有字符的集合称为字符集。通常，我们称 ISO/Unicode 所定义的字符集为 Unicode。在 Unicode 中，每个字符占据一个码位（Code point）。Unicode 字符集总共定义了 1 114 112 个这样的码位，使用从 0 到 10FFFF 的十六进制数唯一地表示所有的字符。不过不得不提的是，虽然字符集中的码位唯一，但由于计算机存储数据通常是以字节为单位的，而且出于兼容之前的 ASCII、大数小段数段、节省存储空间等诸多原因，通常情况下，我们需要一种具体的编码方式来对字符码位进行存储。比较常见的基于 Unicode 字符集的编码方式有 UTF-8、UTF-16 及 UTF-32（一般人常常把 UTF-16 和 Unicode 混为一谈，在阅读各种资料的时候读者要注意区别）。

以 UTF-8 为例，其采用了 1~6 字节的变长编码方式编码 Unicode，英文通常使用 1 字节表示，且与 ASCII 是兼容的，而中文常用 3 字节进行表示。UTF-8 编码由于较为节约存储空间，因此使用得比较广泛。表 8-1 所示就是 UTF-8 的编码方式。

表 8-1　UTF-8 的编码方式

Unicode 符号范围（十六进制）	UTF-8 编码方式（二进制）
0000 0000—0000 007F	0xxxxxxx
0000 0080—0000 07FF	110xxxxx 10xxxxxx
0000 0800—0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
0001 0000—0010 FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

注意　事实上，现行桌面系统中，Windows 内部采用了 UTF-16 的编码方式，而 Mac OS、Linux 等则采用了 UTF-8 编码方式。

除了基于 Unicode 字符集的 UTF-8、UTF-16 等编码外，在中文语言地区，我们还有一些常见的字符集及其编码方式，GB2312、Big5 就是其中影响最大、使用最广泛的两种。

GB2312 的出现先于 Unicode。早在 20 世纪 80 年代，GB2312 作为简体中文的国家标准被颁布使用。GB2312 字符集收入 6763 个汉字和 682 个非汉字图形字符，而在编码上，是采用了基于区位码的一种编码方式，采用 2 字节表示一个中文字符。GB2312 在中国大陆地区及新加坡都有广泛的使用。

而 BIG5 则常见于繁体中文，俗称“大五码”。BIG5 是长期以来的繁体中文的业界标准，共收录了 13 060 个中文字，也采用了 2 字节的方式来表示繁体中文。BIG5 在中国台湾、香港、澳门等地区有着广泛的使用。

扩展　关于内码和交换码

内码实际就是字符在计算机存储单元中的二进制表示，在早期中文字符编码混乱的时候，内码和交换码等概念就产生了。每一种二进制表示的中文的编码都被认为是一种内码，而在有多种内码的情况下，交换码被设计为协调不同的内码间数据交换的手段。依照这种认知方式，UTF-8 等编码即是内码，也是交换码。随着时代的发展和各种标准的先后制定，内码和交换码的概念也在被逐渐淡化，因为通常情况下，两者总是一致的。

不同的编码方式对于相同的二进制字符串的解释是不同的。常见的，如果一个 UTF-8 编码的网页中的字符串按照 GB2312 编码进行显示，就会出现乱码。而 BIG5 和 GB2312 之间的乱码则在中文地区软件中有着“悠久”的历史。不过随着 Unicode 的使用和发展，以及软件系统对多种编码的支持，程序发生乱码的现象也越来越少。总的说来，Unicode 还在其发展期，Unicode、GB2312 以及 BIG5 等多种编码共存的状况可能在以后较长的时间内都会持

续下去。

8.3.2 C++11 中的 Unicode 支持

在 C++98 标准中，为了支持 Unicode，定义了“宽字符”的内置类型 wchar_t。不过不久程序员便发现 C++ 标准对 wchar_t 的“宽度”显然太过容忍，在 Windows 上，多数 wchar_t 被实现为 16 位宽，而在 Linux 上，则被实现为 32 位。事实上，C++98 标准定义中，wchar_t 的宽度是由编译器实现决定的。理论上，wchar_t 的长度可以是 8 位、16 位或者 32 位。这样带来的最大的问题是，程序员写出的包含 wchar_t 的代码通常不可移植。

这一状况在 C++11 中得到了一定的改善，至少 C++11 解决了 Unicode 类型数据的存储问题。C++11 引入以下两种新的内置数据类型来存储不同编码长度的 Unicode 数据。

- char16_t：用于存储 UTF-16 编码的 Unicode 数据。
- char32_t：用于存储 UTF-32 编码的 Unicode 数据。

至于 UTF-8 编码的 Unicode 数据，C++11 还是使用 8 字节宽度的 char 类型的数组来保存。而 char16_t 和 char32_t 的长度则犹如其名称所显示的那样，长度分别为 16 字节和 32 字节，对任何编译器或者系统都是一样的。

此外，C++11 还定义了一些常量字符串的前缀。在声明常量字符串的时候，这些前缀声明可以让编译器使字符串按照前缀类型产生数据。事实上，C++11 一共定义了 3 种这样的前缀：

- u8 表示为 UTF-8 编码。
- u 表示为 UTF-16 编码。
- U 表示为 UTF-32 编码。

3 种前缀对应于 3 种不同的 Unicode 编码。一旦声明了这些前缀，编译器会在产生代码的时候按照相应的编码方式存储。以上 3 种前缀加上基于宽字符 wchar_t 的前缀“L”，及不加前缀的普通字符串字面量，算来在 C++11 中，一共有了 5 种方式来声明字符串字面量，其中 4 种是前缀表达的。

通常情况下，按照 C/C++ 的规则，连续在代码中声明多个字符串字面量，则编译器会自动将其连接起来。比如 "a" "b" 这样声明的方式与 "ab" 的声明方式毫无区别。而一旦连续声明的多个字符串字面量中的某一个是前缀的，则不带前缀的字符串字面量会被认为与带前缀的字符串字面量是同类型的。比如声明 u"a" "b" 和 "a" u"b"，其效果跟 u"ab" 是完全等同的，都是生成了连续的字面量等于 UTF-16 编码 "ab" 的字符串。不过最好不要将各种前缀字符串字面量连续声明，因为标准定义除了 UTF-8 和宽字符字符串字面量同时声明会冲突外，其他字符串字面量的组合最终会产生什么结果，以及会按照什么类型解释，是由编译器实现自行决定的。因此应该尽量避免这种不可移植的字符串字面量声明方式。

对于 Unicode 编码字符的书写，C++11 中还规定了一些简明的方式，即在字符串中用 '\u' 加 4 个十六进制数编码的 Unicode 码位（UTF-16）来标识一个 Unicode 字符。比如 '\u4F60' 表示的就是 Unicode 中的中文字符“你”，而 '\u597D' 则是 Unicode 中的“好”。此外，也可以通过 '\U' 后跟 8 个十六进制数编码的 Unicode 码位（UTF-32）的方式来书写 Unicode 字面常量。程序员获得 Unicode 码位的编码的方法很多，比如在 Windows 系统下，可以使用系统自带的字符映射表，而在网络上，也可以轻松地找到很多免费提供的中文到 Unicode 的在线转换服务的网站。

我们可以来看一下代码清单 8-14 所示的这个例子。

代码清单 8-14

```
#include <iostream>
using namespace std;

int main(){
    char utf8[] = u8"\u4F60\u597D\u554A";
    char16_t utf16[] = u"hello";
    char32_t utf32[] = U"hello equals \u4F60\u597D\u554A";

    cout << utf8 << endl;
    cout << utf16 << endl;
    cout << utf32 << endl;

    char32_t u2[] = u"hello";    // Error
    char u3[] = U"hello";        // Error
    char16_t u4 = u8"hello";     // Error
}

// 编译选项 :clang++ 8-3-1.cpp -std=c++11
```

在本例中，我们声明了 3 种不同类型的 Unicode 字符串 utf8、utf16 和 utf32。由于无论对哪种 Unicode 编码，英文的 Unicode 码位都相同，因此只有非英文使用了 "\u" 的码位方式来标志。我们可以看到，一旦使用了 Unicode 字符串前缀，这个字符串的类型就确定了，仅能放在相应类型的数组中。u2、u3、u4 就是因为类型不匹配而不能通过编译。

如果我们注释掉不能通过的定义，编译并运行代码清单 8-14，在我们的实验机上可以得到以下输出：

```
你好啊
0x7fffaf087390
0x7fffaf087340
```

对应于 char utf8[] = u8"\u4F60\u597D\u554A" 这句，该 UTF-8 字符串对应的中文是“你好啊”。而对于 utf16 和 utf32 变量，我们本来期望它们分别输出“hello”及“hello equals 你

好啊”。不过实验机上我们都只得到了一串数字输出。这是什么原因呢?

事实上，C++11 虽然在语言层面对 Unicode 进行了支持，但语言层面并不是唯一的决定因素。用户要在自己的系统上看到正确的 Unicode 文字，还需要输出环境、编译器，甚至是代码编辑器等的支持。我们可以按照编写代码、编译、运行的顺序来看看它们对整个 Unicode 字符串输出的影响。

首先会影响 Unicode 正确性的过程是源文件的保存。以字符 "\u4F60" 为例，其保证的是输入数据等同于 Unicode 中码位为 4F60 的字符，而被保存的源代码文件中，数据采用的编码则跟编辑器有关。如编辑器采用了 GB2312 编码保存数据，则源代码文件中 utf8 变量的前 2 字节保存的是 GB2312 编码的中文字“你”。而如果编辑器采用了 UTF-8 编码，则源代码文件中的 utf8 变量的前 3 字节保存的是 UTF-8 的中文字“你”。

第二个会影响 Unicode 正确性的过程是编译。C++11 中的 u8 前缀保证编译器把 utf8 变量中的数据以 UTF-8 的形式产生在目标代码的数据段中。不过通常编译器也会有自己的设定，如果编译器被设置了正确的编码形式，(比如文件保存为 GB2312 编码，编译器也设置了文件格式为 GB2312，或者两者均为 UTF-8)，则 u8 前缀能够正常工作。

第三个会影响 Unicode 正确性的过程是输出。C++ 的操作符“<<”保证把数据以字节(char)、双字(char16_t)、四字(char32_t)的方式输出到输出设备，但输出设备(比如在 Linux 下的 shell，或是 Windows 下的 console)是否能够支持该编码类型的输出，则取决于设备驱动等软件层。

我们的实验机是一台 Linux 机器。对于 Linux 而言，大多数软件如 shell、编辑器 vi，以及编译器 g++ 等都会根据 Linux 系统 locale 设定而采用 UTF-8 编码。在代码清单 8-14 所示的例子中，utf8 变量会输出正确，而 utf16、utf32 数据输出均失败，原因就是因为系统并不支持 UTF-16 和 UTF-32 输出。

在现有的编程环境支持下，如果要保证在程序中直接输入中文得到正确的输出，我们建议程序员要使用与系统环境中相同的编码方式。比如在 Linux 下(现在很多 Linux 系统的发布版均使直接用 UTF-8 作为系统中的编码)，u8 前缀的 UTF-8 编码 Unicode 会得到广泛的支持。而 Windows 由于内部采用了 UTF-16 的方式保存文字编码，因此 u 前缀的 UTF-16 编码的 Unicode 可能会被支持得更好。而如果程序员想在不同系统下编译相同的文件(这也并不少见，比如在一些基于 QT IDE 的跨平台开发上，程序员会在各平台间共享源代码)，程序员则应该注意查看编辑器与编译器是否使用了不同的编码方式，并按需调整。

如果在用户确认了使用环境没有问题，在程序员排除了上述环境上的困难之后，又有了 char16_t、char32_t 以及各种前缀表示、\u 字面值等，是否意味着 Unicode 真的就可以良好运作了呢?让我们来看看代码清单 8-15 所示的这个的例子。

代码清单 8-15

```
#include <iostream>
using namespace std;

int main() {
    char utf8[] = u8"\u4F60\u597D\u554A";
    char16_t utf16[] = u"\u4F60\u597D\u554A";


    cout << sizeof(utf8) << endl;        // 10 字节
    cout << sizeof(utf16) << endl;       // 8 字节

    cout << utf8[1] << endl;     // 输出不可见字符
    cout << utf16[1] << endl;    // 输出 22909(0x597D)
}

// 编译选项 :g++ -std=c++118-3-2.cpp
```

这个例子里，我们首先看不同编码情况下 Unicode 字符串的大小。可以看到，UTF-8 由于采用了变长编码，在这里把每个中文字符编码为 3 字节，再加上 '\0' 的字符串终止符，所以 utf8 变量的大小是 10 字节。而 UTF-16 则是定长编码，所以 utf16 占用了 8 字节空间。倘若我们按照使用 ASCII 字符的思路来使用 Unicode 字符，比如使用数组来访问的时候，我们发现 utf8 的输出是不正确的（这里的 utf16 是正确的，只是实验机无法正常输出）。事实上，我们将 UTF-8 编码的数据放在了一个 char 类型中，所以 utf8[1] 只是指向了第一个 UTF-8 字符 3 字节中的第二位，因此输出不正常。

相比于定长编码的 UTF-16，变长编码的 UTF-8 的优势在于支持更多的 Unicode 码位，而且也没有大数端小端段问题（而有字节序问题的 UTF-16 有 LE 和 BE 两种不同版本）。不过不能直接数组式访问是 UTF-8 的最大的缺点。此外，C++11 为 char16_t 和 char32_t 分别配备了 u16string 及 u32string 等字符串类型，却没有 u8string（因为从实现上讲，变长的 UTF-8 编码的数据也不是很容易与 string 配合使用）。这样一来，UTF-8 的字符串不能够被方便地进行增删、查找，至于利用各种高级的 STL 算法，就更加困难了。

倘若用户要完成上面的各种复杂的操作，需要的是一个复杂的类型，比如说用 utf8_t 的类型来保存变长的 UTF-8 字符，而不是像现在这样用 char 数组来“存放”UTF-8 字符。这个想法固然也有一些道理，但 utf8_t 类型给 C++ 带来的冲击可能也是很大的，因为它看起来像是个基本类型，却是变长的，与已有算法结合并不一定有性能上的优势（比如计算第 N 个元素的时间复杂度不再是 O(1)）。

UTF-8 变长的设定更多时候是为了在序列化时节省存储空间，定长的 UTF-16 编码或者 UTF-32 则更适合在内存环境中操作。因此，在现有 C++ 编程中，总是倾向于在 I/O 读写的时候才采用 UTF-8 编码，即在进行 I/O 操作时才将定长 Unicode 编码转化为 UTF-8 使用。内

存中一直操作的是定长的 Unicode 编码，故不过在这种使用方式下，编码转换就成了更加常用且不可或缺的功能。

8.3.3 关于 Unicode 的库支持

C++11 在标准库中增加了一些 Unicode 编码转换的支持。由于 char16_t 及 char32_t 也是 C11 标准中新增的类型，所以 C 库及 C++ 库均有一些不同的实现。

首先我们可以看一些比较直观的编码转换函数。在 C11 中，程序员可以使用库中的一些新增的编码转换函数来完成各种 Unicode 编码间的转换。函数的原型如下：

```
size_t mbrtoc16(char16_t * pc16, const char * s, size_t n, mbstate_t * ps);
size_t c16rtomb(char * s, char16_t c16, mbstate _t * ps);
size_t mbrtoc32(char32_t * pc32, const char * s, size_t n, mbstate_t * ps);
size_t c32rtomb(char * s, char32_t c32, mbstate_t * ps);
```

上述代码中，字母 mb 是 multi-byte（这里指多字节字符串，后面会解释）的缩写，c16 和 c32 则是 char16 和 char32 的缩写，rt 是 convert（转换）的缩写。代码中的几个函数原型大同小异，目的就是完成多字节字符串、UTF-16 及 UTF-32 之间的一些转换。除了 mbstate_t 是用于返回转换中的状态信息外，其余部分意义比较明显，读者应该能直观理解它们的含义。代码清单 8-16 所示是一个可能通过编译的例子。

代码清单 8-16

```
#include <iostream>
#include <cuchar>
using namespace std;

int main() {
    char16_t utf16[] = u"\u4F60\u597D\u554A";
    char mbr[sizeof(utf16)*2] = {0};     // 这里我们假设 buffer 这么大就够了
    mbstate_t s;

    c16rtomb(mbr, utf16, &s);
    cout << mbr << endl;
}
// 编译选项 :g++ -std=c++11 -c 8-3-3.cpp
```

使用 C11 中编码转换函数需要 include 头文件 <cuchar>。不过在本书写作的时候，我们使用的编译器都还没能提供这个头文件及其实现。所以代码清单 8-16 所示的例子仅供参考。

C++ 对字符转换的支持则稍微复杂一点，不过 C++ 对编码转换支持的新方法都需要源自于 C++ 的 locale 机制的支持[⊖]。事实上，locale 的概念在 POSIX 中就用，在 C++ 中，通常情况下，locale 描述的是一些必须知道的区域特征，如程序运行的国家 / 地区的数字符号、日期

⊖ 可以参考该文理解 C++ 的 locale 机制：http://www.cantrip.org/locale.html。

表示、钱币符号等。比如在美国地区且采用了英文和 UTF-8 编码，这样的 locale 可以表示为 en_US.UTF-8，而在中国使用简体中文并采用 GB2312 文字编码的 locale 则可以被表示为 zh_CN.GB2312，等等。

通常知道了一个地区的 locale，要使用不同的地区特征，则需访问该 locale 的一个 facet。facet 可以简单地理解为是 locale 的一些接口。比如对于所有的 locale 都会有 num_put/num_get 的操作，那么这些操作就是针对该 locale 数值存取的接口，即该 locale 情况下数值存取的 facet。在 C++ 中常见的 facet 除去 num_get/num_put、money_get/money_put 等外，还有一种就是 codecvt。

codecvt 从类型上来讲是一个模板类，从功能上讲，是一种能够完成从当前 locale 下多字符编码字符串到多种 Unicode 字符编码转换（也包括 Unicode 字符编码间的转换）的 facet。这里的多字节字符串不仅可以是 UTF-8，也可以是 GB2312 或者其他，其实际依赖于 locale 所采用的编码方式。在 C++ 标准中，规定一共需要实现 4 种这样的 codecvt facet ⊖。

```
std::codecvt<char, char, std::mbstate_t>          // 完成多字节与 char 之间的转换
std::codecvt<char16_t, char, std::mbstate_t>      // 完成 UTF-16 与 UTF-8 间的转换
std::codecvt<char32_t, char, std::mbstate_t>      // 完成 UTF-32 与 UTF-8 间的转换
std::codecvt<wchar_t, char, std::mbstate_t>       // 完成多字节与 wchar_t 之间的转换
```

每种 facet 负责不同类型编码数据的转换。值得注意的是，现行编译器支持情况下，一种 locale 并不一定支持所有的 codecvt 的 facet。程序员可以通过 has_facet 来查询该 locale 在本机上的支持情况，如代码清单 8-17 所示。

代码清单 8-17

```
#include <iostream>
#include <locale>
using namespace std;

int main(){
    // 定义一个 locale 并查询该 locale 是否支持一些 facet
    locale lc("en_US.UTF-8");
    bool can_cvt = has_facet<codecvt<wchar_t, char, mbstate_t>>(lc);
    if (!can_cvt)
        cout << "Do not support char-wchar_t facet!" << endl;

    can_cvt = has_facet<codecvt<char16_t, char, mbstate_t>>(lc);
    if (!can_cvt)
        cout << "Do not support char-char16 facet!" << endl;

    can_cvt = has_facet<codecvt<char32_t, char, mbstate_t>>(lc);
    if (!can_cvt)
        cout << "Do not support char-char32 facet!" << endl;
```

⊖ 参见 http://en.cppreference.com/w/cpp/locale/codecvt。

```
    can_cvt = has_facet<codecvt<char, char, mbstate_t>>(lc);
    if (!can_cvt)
        cout << "Do not support char-char facet!" << endl;

    return 0;
}

// 编译选项 :g++ -std=c++11 8-3-4.cpp
```

编译运行代码清单 8-17，在我们的实验机环境及编译器支持情况下，可以得到以下结果：

```
Do not support char-char16 facet!
Do not support char-char32 facet!
```

由上述结果可知，从 char 到 char16 或 char32 转换的两种 facet 还没有被支持（实验机使用的编译器尚未支持）。

而在使用 facet 上，用户并不需要显式地在代码中生成 codecvt 对象。比如在对 C++11 中 stream 进行 I/O 时，我们只需要一些简单的设定，就可以让 stream 自动进行一些编码的转换。我们看一下代码清单 8-18 所示的例子[⊖]。

代码清单 8-18

```
#include <iostream>
#include <fstream>
#include <string>
#include <locale>
#include <iomanip>
using namespace std;

int main()
{
    // UTF-8 字符串 , "\x7a\xc3\x9f\xe6\xb0\xb4\xf0\x9d\x84\x8b";
    ofstream("text.txt") << u8"z\u00df\u6c34\U0001d10b";

    wifstream fin("text.txt");
    // 该 locale 的 facet - codecvt<wchar_t, char, mbstate_t>
    // 可以将 UTF-8 转化为 UTF-32
    fin.imbue(locale("en_US.UTF-8"));

    cout << "The UTF-8 file contains the following wide characters: \n";
    for(wchar_t c; fin >> c; )
        cout << "U+" << hex << setw(4) << setfill('0') << c << '\n';
}

// 编译选项 :g++ -std=c++11 8-3-5.cpp
```

⊖ 本例来源于 http://en.cppreference.com/w/cpp/locale/codecvt，仅做了注释上的修改。

在代码清单 8-18 中，我们使用了 wifstream 来打开一个 UTF-8 编码的文件。随后调用了这个 wifstream 的 imbue 函数，为其设定了一个为 en_US.UTF-8 的 locale。这样一来当进行 I/O 操作的时候，会使用完成 UTF-8 到 UTF-32 编码转换的 facet（codecvt<wchar_t, char, mbstate_t>）来完成编码转换。编译运行代码清单 8-18，我们就可以看到定义的 Unicode 字符串的十六进制表示。

```
The UTF-8 file contains the following wide characters:
U+007a
U+00df
U+6c34
U+1d10b
```

codecvt 还派生一些形如 codecvt_utf8、codecvt_utf16、codecvt_utf8_utf16 等可以用于字符串转换的模板类。这些模板类配合 C++11 定义的 wstirng_convert 模板，可以进行一些不同字符串的转换。代码清单 8-19 也是一个 C++11 标准中的示例，不过由于我们编译器尚未支持，所以也仅供参考。

代码清单 8-19

```
#include <cvt/wstring>
#include <codecvt>
#include <iostream>
using namespace std;
int main() {
    wstring_convert<codecvt_utf8<wchar_t>> myconv;
    string mbstring = myconv.to_bytes(L"Hello\n");
    cout << mbstring;
}
```

除了 to_bytes 外，wstring_convert 还支持使用 from_bytes 来完成逆向的编码转换。更多关于 wstring_convert、locale、codecvt 的内容，读者可以参看一些在线文档，这里不再展开描述。

此外，还有一点值得注意，在 C++98 标准定义 wchar_t 类型的时候，为其添加了新的 fstream 类型，如 wifstream 及 wofstream 等。不过 C++11 标准并没有为 char16_t 及 char32_t 再次产生 fstream 对象。关于这点，跟前面提到的 UTF-8 操作问题有类似。标准委员会意识到在 Unicode 在序列化存储时很少是 UTF-16 或者是 UTF-32 的（空间太过浪费）。所以从实际情况出发，程序员可以利用不同的 codecvt 的 facet 来将 UTF-8 编码存储的字符与不同的 Unicode 进行转换，而不必直接将 UTF-16 和 UTF-32 编码的字符存储到文件，基于此，也就没在 C++11 标准中提供支持该功能的 u16ifstream、u32ofstream 等。

事实上，尽管 C++11 对 Unicode 做了更多的支持，Unicode 字符串的使用仍然比 ASCII 字符复杂。如我们所见的，程序在进行各种 I/O 操作时，往往需要 UTF-8 编码的字符。程序

员如果想直接在内存中操作 UTF-8 编码字符，那么对 UTF-8 字符串的 string 进行遍历、插入、删除、查找等操作会比较困难。如果遇到这样的情况，程序员可以自行寻求一些第三方库的支持。

8.4 原生字符串字面量

☞类别：所有人

原生字符串字面量（raw string literal）并不是一个新鲜的概念，在许多编程语言中，我们都可以看到对原生字符串字面量的支持。原生字符串使用户书写的字符串“所见即所得”，不再需要如 '\t'、'\n' 等控制字符来调整字符串中的格式，这对编程语言的学习和使用都是具有积极意义的。

顺应这个潮流，在 C++11 中，终于引入了原生字符串字面量的支持。C++11 中原生字符串的声明相当简单，程序员只需要在字符串前加入前缀，即字母 R，并在引号中用使用括号左右标识，就可以声明该字符串字面量为原生字符串了。请看下面的例子，如代码清单 8-20 所示。

代码清单 8-20

```
#include <iostream>
using namespace std;

int main(){
    cout << R"(hello,\n
         world)" << endl;
    return 0;
}


// 编译选项 : g++ 8-1-2.cpp -std=c++11
```

代码清单 8-20 的输出如下，可以看到 '\n' 并没有被解释为换行。

```
hello,\n
         world
```

而对于 Unicode 的字符串，也可以通过相同的方式声明。声明 UTF-8、UTF-16、UTF-32 的原生字符串字面量，将其前缀分别设为 u8R、uR、UR 就可以了。不过有一点需要注意，使用了原生字符串的话，转义字符就不能使用了，这会给想使用 \u 或者 \U 的方式写 Unicode 字符的程序员带来一定影响。下面来看代码清单 8-21 所示的例子。

代码清单 8-21

```
#include <iostream>
```

```
using namespace std;

int main(){
    cout << u8R"(\u4F60,\n
         \u597D)" << endl;
    cout << u8R"(你好)" << endl;
    cout << sizeof(u8R"(hello)") << "\t" << u8R"(hello)" << endl;
    cout << sizeof(uR"(hello)") << "\t" << uR"(hello)" << endl;
    cout << sizeof(UR"(hello)") << "\t" << UR"(hello)" << endl;
    return 0;
}

// 编译选项:g++ -std=c++11 8-4-2.cpp
```

编译运行代码清单 8-21，可以得到以下结果：

```
\u4F60,\n
         \u597D
你好
6       hello
12      0x400be6
24      0x400bf4
```

可以看到，当程序员试图使用 \u 将数字转义为 Unicode 的时候，原生字符串会保持程序员所写的字面值，所以这样的企图并不能如愿以偿。而借助文本编辑器直接输入中文字符，反而可以在实验机的环境下在文件中有效地保存 UTF-8 的字符（因为编辑器按照 UTF-8 编码保存了文件）。程序员应该注意到编辑器使用的编码对 Unicode 的影响。而在之后面的 sizeof 运算符中，我们看到了不同编码下原生字符串字面量的大小，跟其声明的类型是完全一致的。

此外，原生字符串字面量也像 C 的字符串字面量一样遵从连接规则。我们可以看看代码清单 8-22 所示的例子。

代码清单 8-22

```
#include <iostream>
using namespace std;

int main() {
    char u8string[] = u8R"(你好)" " = hello";
    cout << u8string << endl;    // 输出"你好 = hello"
    cout << sizeof(u8string) << endl;    // 15
    return 0;
}

// 编译选项:g++ -std=c++11 8-4-3.cpp
```

可以看到，代码清单 8-22 中的原生字符串字面量和普通的字符串字面量会被编译器自动连接起来。整个字符串有 2 个 3 字节的中文字符，以及 8 个 ASCII 字符，加上自动生成的 \0，字符串的总长度为 15 字节。与非原生字符串字面量一样，连接不同前缀的（编码）的字符串有可能导致不可知的结果，所以程序员总是应该避免这样使用字符串。

8.5 本章小结

本章中我们了解了 C++11 支持的 4 种新特性：对齐方式、通用属性、Unicode，以及原生字符串字面量。

对齐方式本是语言设计者想掩藏的细节，不过在 C++11 编程方式越发复杂的情况下，提供给用户更底层的手段往往是必不可少的。在一些情况下，用户虽然不能保证总是写出平台无关，或者说各平台性能最优的代码，但只需改造 alignas 之后的对齐值参数就可以保证程序的移植性及性能良好，也不失为一种好的选择。而 C++11 对对齐方式的支持从语法规则到库，基本上考虑到了各种情况，可以说是相当完备的。

而通用属性则像是关键字的包装器。一度有人认为，C++ 应该是用通用属性而不是关键字来实现一些特征，不过最后的结论却是：语言本身的所有特性都应该是关键字，通用属性仅仅用在不改变语义的场合，比如产生编译警告、优化提示等。从现在的情况看来，通用属性的语法规则意义大于现在已有的两个预定义通用属性。编译器厂商或组织或者标准委员会在对语言进行扩展的时候，可能还会利用这样的通用属性的语法规则。

C++11 还增强了对 Unicode 的支持。针对以前长度并不明确的 wchar_t，增加了 char16_t 及 char32_t 两种内置类型。考虑到变长编码 UTF-8 使用上的不方便，以及定长的 UTF-16 和 UTF-32 在存储或者一些其他方面的弱势，C++11 在逐步加强对 Unicode 类型转换方面的支持。不过基于 Unicode 的编程是否容易了很多，可移植性是否加强了很多，可能还需要各位读者慢慢体会。此外，C++11 还支持了原生字符串字面量。这是一个在其他较晚发明的语言中常见的特性，C++11 将其引入其中，也算方便了程序员对 C++ 字符串的学习和使用。

附录 A

C++11 对其他标准的不兼容项目

在附录部分，我们会详细描述C++11的不兼容性（incompatibility）、废弃的特性（deprecated feature），以及编译器支持状况（compiler support status）。虽然这些内容不及第2 ~ 8章的“核心”内容重要，不过却常常具有很高的实用性。我们建议读者可以粗略地阅读一遍相关内容。这样在遇到一些实际编程问题的时候，读者就可能理解问题的来由。这几个附录的内容上可能存在着一些重复，不过，我们还是保持了这样的重复，以保证从每个视角出发的描述的完整性。

那么本附录要讲解的是C++11的一些不兼容性。虽然C++11作为C/C++的“嫡传后裔”，对C/C++98/03做到了最大的兼容，不过一些显著的不兼容性还是存在的，我们可以分别通过比较C++11与C++03、C++与ISO C，以及比较C++11与C11来进行了解。

A.1 C++11和C++03的不兼容项目

条目1 在C++11中R、u8、u8R、u、uR、U、UR和LR是新的字符串修饰符，当用它们来修饰字符串时，即使它们是宏名，也将作为修饰符来解释。比如：

```
#define u8 "AAAAA"
const char * s = u8"u-eight-string";
```

在C++03中s是字符串“AAAAAu-eight-string”；在C++11中s是一个UTF-8的字符串，其内容是“u-eight-string”。

条目2 C++11支持用户自定义的字面常量，这会引起一些和C++03不一致的行为，比如：

```
#define _x " world"
"hello"_x
```

在C++03中"hello"_x会拼接成hello world；而在C++11中"hello"_x会作为一个用户自定义字面常量来使用，例如：

```
std::string operator ""_x(const char* s) {
   return std::string(s);
}
```

那么"hello"_x将会作为函数调用返回一个类型为std::string的变量，这个返回变量的内

容是 hello。

条目 3 C++11 引入了一些新的关键字，如果 C++03 代码用到这些标识符会被 C++11 视为非法的代码。这些关键字包括 alignas、alignof、char16_t、char32_t、constexpr、decltype、noexcept、nullptr、static_assert 和 thread_local。

条目 4 C++11 引入了 C99 的新类型 long long。类似 C99，对于长于 long 类型的整型常量将会被转换成 signed long long 类型。而在 C++03 中，长于 long 类型的整型常量将会被转换成无符号整数，如 unsigned long。例如 214748364700 在 C++11 中将会被识别为 long long 类型数据。

条目 5 C++11 和 C99 一样，对整数向“0”取商（/）或取余（%）；而 C++03 允许向负无穷取商或取余。

条目 6 关键字 auto 不再被用来作为存储类型的修饰符，而其表示修饰的类型是由初始化表达式推导而来。

条目 7 C++11 要求数组初始化时，不能将数据的类型收窄。下面的代码在 C++03 中合法，而在 C++11 中非法。

```
int arr[] = {1.0};
```

这里 1.0 是一个 double 类型，使用它初始化 int 类型数组会导致数据收窄。因此在 C++11 中无法通过编译。

条目 8 可能导致问题的隐式函数在 C++11 被定义为 deleted。这些隐式函数不能被使用。而 C++03 中可以使用。比如说下面这段代码：

```
struct A{ const int a; };
```

由于常量（const）a 总是应该被静态初始化的，因此程序员应该为 struct A 提供一个构造函数来完成这样的初始化。在 C++11 中，遇到这种可能导致问题（would be ill-formed）的情况下，缺省构造函数将被删除，以提示用户可能存在问题。

条目 9 C++11 中去除了无用的关键字：export。

条目 10 C++11 中，模板嵌套时可以直接使用双右尖括号，C++03 则需要空白字符填充尖括号。例如：

```
template <typename T> struct X { };
template <int N> struct Y { };
X< Y< 1 >> 2 > > x;
```

C++03 中会解释为 X< Y< (1 >> 2) > > x; ==> X< Y< 0 > > x。

C++11 中这是非法的声明，因为“X< Y< 1 >>”被视为有效的模板使用。

条目 11 C++11 引入了一些新的标准头文件：<array>、<atomic>、<chrono>、<codecvt>、<condition_variable>、<forward_list>、<future>、<initializer_list>、<mutex>、<random>、<ratio>、<regex>、<scoped_allocator>、<system_error>、<thread>、<tuple>、<typeindex>、<type_traits>、<unordered_map>、<unordered_set>。

还有一些新加入的和 C 兼容的头文件：<ccomplex>、<cfenv>、<cinttypes>、<cstdalign>、<cstdbool>、<cstding>、<ctgmath>、<cuchar>。

条目 12 C++11 中，swap 方法从 <algorithm> 移到了 <utility> 中。

条目 13 C++11 加入了一个新的顶级 namespace：posix。

条目 14 通用属性中的标记符如 carries_dependency、noreturn 不能作为宏名。

条目 15 C++03 假设全局的 new 操作符只会抛出类型为 std::bad_alloc 的异常；而 C++11 允许全局的 new 操作符抛出其他类型的异常。

条目 16 C++11 要求 errno 变量是线程局部的，而不是全局的。

条目 17 C++11 支持轻量级的垃圾回收机制。

条目 18 标准库中的仿函数（函数对象）不再继承自 std::unary_function 和 std::binary_function。

条目 19 标准容器要提供的 size() 成员函数要求是 O(0) 复杂度；C++03 中 std::list 的成员 size() 允许线性复杂度。

条目 20 C++11 中改变了一些函数方法的原型，比如 erase 和 insert 的返回值类型 iterator 变成了 const_iterator，resize 函数的参数从传值改为了传引用。

条目 21 C++11 允许一些类和函数方法的实现不同于 C++03，比如：std::remove、std::remove_if、std::complex、std::ios_base::failure。

A.2 C++ 和 ISO C 标准的不兼容项目

条目 1 C++ 中很多关键字是 C 所没有的（不详细列举）。

条目 2 C 中字符常量的类型是 int，C++ 中是 char。如果在 C++ 代码中同时有以下两个版本的 f 函数的定义：

```
void f (char c);
void f(int);
```

那么，函数调用 f('x') 会选择 void f (char c) 版本。

条目 3 C++ 中字符串常量的类型是 const char[]，而 C 中字符串常量的类型是 char[]。

条目 4 C 中允许文件范围中的变量重复定义。如：

```
int var;
int var;
```

在 C 中是允许的；而这在 C++ 中是不允许的。

条目 5 不带 extern 关键字的 const 变量在 C++ 中是 internal linkage（内部链接的，即不可以被其他文件中同名变量引用）；而在 C 中则是 external liankage（外部链接的，即可以被其他文件中的同名变量引用）。

条目 6 C++ 要求从 void* 类型变量到其他类型的转化必须是显式的；而 C 中则不需要显式转换。

条目 7 C++ 中只有非常量非易变对象（non-const ,non-volatile）指针可以转换为 void* 类型。

条目 8 C++ 不接受隐式声明函数。这个特性在 ISO C 中也逐渐被抛弃，比如：

```
int main(){ printf("hello\n");}
```

这里因为 printf 没有定义（没有 #include <stdio.h>），C++ 会编译时报错 printf 未声明；而 C 会把 printf 当作一个 int printf（任意参数）的函数类型。如果运行时动态库中不存在 printf 这个函数的话，则会导致运行时错误。

条目 9 不能在结构体或类型的声明上加 static 关键字。比如：

```
static struct st { int i; };
```

在 C 中 static 关键字将被忽略；C++ 中这则是错误的语法。

条目 10 C++ 中 typedef 的类型别名不能和已有的类型同名。

条目 11 常量（const）对象在 C++ 中必须初始化；在 C 中则没这个限制。

条目 12 隐式 int 类型在 C++ 中被禁止，C 中也逐渐抛弃。比如：func(){} 这样的声明方式在 C 语言中是可以的；而在 C++ 中，因为 func 没有返回值类型则是非法表达式。

条目 13 关键字 auto 在 C++11 中有新的语义：用于类型自动推导；而 C 中 auto 是修饰对象的存储类型的关键字。

条目 14 C++ 中 enum 对象只能用同类型的 enum 赋值；而 C 中可以用任意的整型数对 enum 变量赋值。C++ 中 enum 变量的类型是对应的 enum 类型；而 C 中 enum 对象的类型是 int 整型。

条目 15 函数声明中的空参数在 C++ 中意味着函数没有参数；而在 C 中则意味着该函数的参数个数未知。

条目 16 C++ 不允许类型定义在函数的参数或返回值类型的位置上；而形如：

```
void f(struct S {int I;} s);
```

这样的表达式在 C 中则可以接受。

条目 17 C++ 不接受老的废弃的函数定义格式：参数在 () 之外，比如：

```
void bar() int par1 {}
```

在 C++ 中就是非法的声明。

条目 18 作用域内部的结构体在 C++ 中会覆盖作用域外部的同名变量，比如：

```
char s;
void f(){
    struct s {
        int i;
    }; // struct s 覆盖 char s
}
```

而在 C 中不会。

条目 19 C++ 中，嵌套的结构体仅在其父结构体作用域中可见；而在 C 中，嵌套的结构体则在全局可见，比如：

```
struct Outter {
    struct Inner {
        int I;
    };
};
```

在 C 中使用 Inner 类型可以直接写：struct Inner in，而 C++ 中使用 Inner 类型则必须带上其父结构体 Outter，则只能写：Out::Inner in。

条目 20 C++ 中 typedef 形成的类型别名不能重定义为其他类型或变量。如下面的代码就是这样一种情况：

```
typedef int Int;
struct S{
   int Int; // 在 C 中合法，而在 C++ 中非法
};
```

条目 21 C++ 中 volatile 的对象不能作为隐式构造函数和隐式赋值函数的参数，比如：

```
struct X { int i; };
volatile struct X x1 = {0};
struct X x2(x1); // 在 C++ 中非法
struct X x3;
x3 = x1; // 在 C++ 中同样非法
```

A.3 C++11 与 C11 的区别

虽然 C11 标准开始起草的时间比 C++0x 晚很多，但 C11 的发布却只比 C++11 晚了几个月。这是因为它们的草案中很多都是相互参考的。因此 C++11 与 C11 的不兼容点并不多。

下面简单地列一些 C++11 和 C11 的特别区别点。

条目 1 C++11 中没有 C11 中支持的 _Generic 关键字，因为 C++ 能够很好地支持重载。

条目 2 C++11 中 noreturn 是一个通用属性。相应地要表示函数永不返回的话，在 C11 中可以使用 _Noreturn 关键字。比如：

```
_Noreturn void outfunc() { abort(); }
```

是 C11 中的表示 outfunc 的方法，它等价于在 C++11 中使用通用属性的 outfunc。

```
[[ noreturn ]] void outfunc() { abort(); }
```

条目 3 许多 C11 的新特性在 C++11 中有对应特性。只是关键字上有一些细微的区别。比如 C++11 中的关键字：

```
alignas, alignof, thread_local, static_assert
```

在 C11 中对应的关键字分别是：

```
_Alignas, _Alignof, _Thread_local,  _Static_assert
```

条目 4 C++11 和 C11 都有 atomic 支持。

C11 中用 _Atomic 修饰符来修饰一个原子数据类型。如：_Atomic int i;

C++11 在 std namespace 下定义了 atomic 模板来支持 atomic 类型，比如：

```
template<class T> struct atomic;
template<> struct atomic<integral>;
template<class T> struct atomic<T*>;
```

条目 5 C11 中用 <threads.h> 中定义的一系列用于互斥操作的函数，比如：mtx_destroy, mtx_init, mtx_lock, mtx_trylock, mtx_unlock 等。

C++11 中通过使用 std namespace 下的一些类：mutex、recursive_mutex 等，通过这些类的成员函数 lock、unlock、trylock 支持互斥操作。

条目 6 C11 用 <threads.h> 中定义的一系列函数来支持线程，如：thrd_create，thrd_current，thrd_detach，thrd_equal，thrd_exit，thrd_join，thrd_sleep，thrd_yield。

C++11 有 thread 类型，该类型有 join、detach 等成员函数。

A.4 针对 C++03 的完善

而谈及兼容性的话，除了上面列举的 C++11 与 C++03、ISO C 以及 C11 的区别外，在 C++11 起草的过程中，也包含了一些对以前标准（C++03）的修改和完善。我们把一些针对 C++03 的完善也列举了出来。

条目 1 .* 和 ->* 操作符的第二个参数不再要求是一个完全类（complete class），即包含了全部声明体的类型。

条目 2 内存释放函数不因抛出异常而终止。

条目 3 C++03 要求，当第一个参数是空指针（null pointer）的时候，内存释放函数（如用户自定义的 delete 操作符）相当于无任何作用。现在不再有这样的限制。

条目 4 如果 typeid 操作符的参数是 cv 修饰的，其结果是对应的无 cv 修饰的类型。

条目 5 在常量表示式中可以使用 throw，如：const char * s = (n == m) ? throw “bad” : “ok”；在 C++11 中是合法的表达式。

在 C++11 中，这样的改进还有很多。更多的信息读者可以参考以下链接：http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html。

附录 B

弃用的特性

随着 C++11 的发布和新特性的出现，一些 C++98 与 C++03 中的特性也即将被淘汰。其中一些被更强大的新特性所取代，如 auto_ptr 等，也有因为各种缺陷而在实际编程中很少被使用的，如 export、register 等。相比于不兼容性，了解为什么弃用往往更能了解语言在如何发展。本章将详细描述并总结被 C++11 所弃用的各种特性。

条目 1　auto 关键字

旧特性：auto 用来标识具有自动存储期的局部变量。

改动：auto 关键字可以用来从变量的初始值中推导出变量的类型。

在旧标准中，auto 用来声明具有自动存储期的局部变量，这样变量就是自动存储类别，属于动态存储方式，当变量离开作用域后存储空间会被自动释放。实际上，所有非静态的局部变量默认都具有自动的存储期，因此 auto 关键字很少被用到。

在 C++11 新标准中，auto 被作为一个新的类型修饰符，声明变量时不用指定变量类型，而是根据该变量的初始化表达式或者一个具有追踪返回类型的函数定义推导得来。下面举一个例子，见以下代码：

```
for (vector<int>::iterator i = vec.begin(); i < vec.end(); i++)  {
   vector<int>::iterator j = i;
}
```

在 C++11 中，我们可以使用 auto 关键字来提高可读性。

```
for (auto i = vec.begin(); i < vec.end(); i++) {
   auto j = i;
}
```

此外，auto 类型推导使用非常灵活，它几乎可以用在任何需要声明变量类型的上下文中，比如命名空间、for 循环的循环变量初始化及 for 循环体，以及判断语句中，甚至能被使用在模板中。但是，auto 不可以声明函数参数，也不能推导数组类型。

由于 auto 在 C++11 中被赋予了新的语义，为了避免混淆，C++11 中 auto 不再作为存储的变量声明，而只作为类型修饰符。

条目 2　语言特性 export

旧特性：用来定义非内联的模板对象和模板函数。

改动：export 特性被移除。export 关键字被保留，但是不包含任何语义。

我们可以使用关键字 extern 来访问其他编译单元中普通类型的变量或对象，而对于模板来说，则需要使用 export 关键字。export 的设计初衷是想要创建一个折中的设计方法，来同时支持模板实例化的包含模型（inclusion model）和独立编译模型（separate compilation model），然而，没有一种增强机制来确保每一种模型的实现都可以很简单。因此，由于实现的难度，很多编译器都没有实现。此外，export 关键字在实际编程过程中也很少被用到。

所以，在 C++11 中，export 关键字的语义被移除。但是 export 仍作为一个无语义的关键字被保留下来。

条目 3 register 关键字（作为存储类）

旧特性：声明将变量存放在寄存器中。

改动：改变为声明存储类的关键字。

在旧标准中，变量用 register 来声明时，表示此变量会被大量用到，因此建议将此变量存放在寄存器中，这样可以提高读取的速度。但是，寄存器的数量是有限的，如果寄存器已满，变量依旧会被存放在存储器中。另一方面，它对于编译器来说只是一种建议，而编译器不一定会执行，实际上，大部分编译器都选择忽略它。因此，register 关键字其实很少被用到，而且，大多数情况下是没有意义的。

在 C++11 新标准中，register 关键字的作用有所改变。用 register 关键字仅能用于一个区块内的变量声明或作为函数参数的声明。它仅仅表示变量拥有自动存储的生命期（像 C++03 中的 auto 一样）。

条目 4 隐式拷贝函数

旧特性：如果类中已经声明了其他拷贝函数或者析构函数，编译器依旧会自动生成一个隐式拷贝函数。

改动：隐式拷贝函数不会自动生成。

在 C++11 中，如果用户已经声明了一个拷贝复制操作符或者一个析构函数，那么编译器不会隐式声明一个拷贝构造函数。同样，如果用户已经声明了一个拷贝构造函数或者析构函数，编译器则不会隐式地声明一个拷贝复制操作符。

条目 5 auto_ptr

旧特性：智能指针，当系统因异常退出时避免资源泄漏。

改动：auto_ptr 被 unique_ptr 所取代。

auto_ptr 类模板中存放了一个指针，它指向一个可以通过 new 得到的对象，并且在此智能指针被析构时向堆归还该对象。这里需要注意的是 auto_ptr 拥有一个严格的所有权机制。auto_ptr 拥有其指针指向的对象的所有权，而复制 auto_ptr 的操作会复制该指针，并将对象的所有权交给目标类。这是为了避免两个 auto_ptr 同时拥有同一个对象，否则程序的行为将是不确定的。

在 C++11 中，unique_ptr 提供了一种比 auto_ptr 更好的解决方案，并取代了 auto_ptr。unique_ptr 是一个对象，它拥有另一个对象，并且能够通过指针来管理它。更准确地说，unique_ptr 对象中有一个指向另一个对象的指针，并且在它自身析构时析构该对象。这些特性都与 auto_ptr 相同。

此外，unique_ptr 也具备了 auto_ptr 的绝大部分特性，除了 auto_ptr 的不安全隐性的左值转移（move）。对于 auto_ptr 来说，拷贝 auto_ptr ，会导致所有权转移，如以下语句：

```
std::auto_ptr<int> a(new int);
std::auto_ptr<int> b = a;
```

同样，拷贝构造函数也会进行所有权的转移，如以下语句：

```
std::auto_ptr<int> c(a);
```

由此可见，auto_ptr 的转移是隐性的，因此程序员可能会在不经意的情况下就把对象转移了，因此是不安全性。

在 unique_ptr 中，要进行对象的转移，需要使用 std::move 函数将对象转换为右值，例如以下语句：

```
std::unique_ptr<int> a(new int);
std::unique_ptr<int> b = std::move(a);
```

这是因为 unique_ptr 对于拷贝行为作了限制。而对于拷贝构造函数来说，unique_ptr 并没有类似以下的构造函数：

```
std::unique_ptr<T>::unique_ptr(std::unique_ptr<T> const&) // 默认 deleted
```

如果在构造时想要复制整数的值，可以用以下语句：

```
std::unique_ptr<int> c(new int(*a));
```

而如果确实想要将 a 中的指针进行转移，则需要调用 std::move：

```
std::unique_ptr<int> d(std::move(a));
```

另外，值得一提的是，unique_ptr 可以存放在标准容器之中。

```
vector<unique_ptr<int>> v;
v.push_back(unique_ptr<int>(new int(0)));
unique_ptr<int> a(new int(0));
v.push_back(move(a));
```

但是，由于 unique_ptr 本身不支持拷贝构造，因此元素类型为 unique_ptr 的容器同样也不支持拷贝构造，这时也需要用到转移构造。

条目 6　bind1st/bind2nd

旧特性：将二元函数对象绑定成一元仿函数（函数对象）。

改动：被 bind 模板所取代。

bind1st 和 bind2nd 函数可以将一个二元函数绑定成一元函数，也就是将二元函数所接受的两个参数之一绑定下来，以此来使函数变成一元的。bind1st 绑定第一个参数，bind2nd 绑定第二个参数。例如：

```
find_if(v.begin(), v.end(), bind2nd(greater<int>(), 5));
```

绑定 greater<int> 的第二个参数为 5，亦即找到向量中第一个大于 5 的整数。

```
find_if(v.begin(), v.end(), bind1st(greater<int>(), 5));
```

绑定 greater<int> 的第一个参数为 5，亦即找到向量中第一个小于 5 的整数。

在 C++11 中，新的 bind 函数模板提供了一种更好的可调用类的参数绑定机制。

```
namespace std {
    template<class T> struct is_bind_expression
    : integral_constant<bool, see below> { };
}
```

接下来我们来看 bind 模板函数，它有如下形式：

```
template<class F, class... BoundArgs>
    unspecified bind(F&& f, BoundArgs&&... bound_args);
```

其中 f 是函数的右值引用，表示要进行绑定的函数对象，BoundArgs 是函数对象的参数类型列表，而 bound_args 是需要绑定的值。如果一个参数需要绑定，那么在调用 bind 函数时传具体参数进去即可，而如果不需要绑定，那么就需要使用占位符，std::placeholders::_J，J 为从 1 开始的正整数。bind 的返回类型为可调用实体，可以直接赋值给 std::function。

如下这个例子：

```
int Func(int x, int y);
function< int(int)> f = bind(Func, 1, placeholders::_1);
f(2); // the same as Func(1, 2);
```

我们可以用 is_bind_expression 来检查由 bind 生成的函数对象，而 bind 也凭借 is_bind_expression 来检查子表达式。对于用户来说，可以借由它来表示在 bind 调用中某个类型应该被当做子表达式来对待。如果 T 是 bind 的返回类型，那么 is_bind_expression 由 integral_

constant<bool, true> 得到，否则由 integral_constant<bool, false> 得到。

另一个值得一提的函数是 is_placeholder，它可以检查标准占位符 _1、_2 等。bind 凭借 is_placeholder 来检查占位符，用户也可以借由此模板来表示占位符类型。如果 T 的类型是 std::placeholders::_J，则 is_placeholder<T> 由 integral_constant<int, J> 得到，否则就由 integral_constant<int, 0> 得到。

由上可以看出 bind 相对于 bind1st 和 bind2nd 来说要灵活得多，它不像 bind1st 和 bind2nd 那样限制原函数对象的参数个数为两个，bind 所接受的函数对象的参数数量没有限制，而且用户可以随意绑定任意个数的参数而不受限制，因此，有了 bind，bind1st 和 bind2nd 明显没有了用武之地而被弃用（deprecated）。

条目 7 函数适配器（adaptor）

旧特性：ptr_fun, mem_fun, mem_fun_ref, unary_function, binary_function

新特性：弃用。

在旧特性中，提供了多个函数适配器。

```
template <class Arg1, class Arg2, class Result>
    pointer_to_binary_function<Arg1,Arg2,Result>
     ptr_fun(Result (*f)(Arg1, Arg2));
```

ptr_fun 的返回值是 pointer_to_binary_function<Arg1,Arg2,Result>(f)。简单来说，它可以将一个函数转化为一个函数对象。以下是一个例子：

```
int compare(const char*, const char*);
    replace_if(v.begin(), v.end(),
    not1(bind2nd(ptr_fun(compare), "abc")), "def");
```

上例将所有 v 序列中的 abc 替换为 def。

另外，ptr_fun 除了可以转化二元函数以外，也可以转化一元函数，此时返回值是 pointer_to_binary_function<Arg,Result>(f)。

```
template <class Arg, class Result>
    pointer_to_unary_function<Arg,Result>
    ptr_fun(Result (*f)(Arg));
```

除了常规函数适配器 ptr_fun 外，还有成员函数适配器 mem_fun 和 mem_fun_ref。

```
template <class S, class T> class mem_fun_t
     : public unary_function<T*, S>
template<class S, class T> mem_fun_t<S,T> mem_fun(S (T::*f)());

template <class S, class T> class mem_fun_ref_t
     : public unary_function<T, S>
```

```
template<class S, class T> mem_fun_ref_t<S,T> mem_fun_ref(S (T::*f)());
```

为了说明 mem_fun 和 mem_fun_ref，看一下以下的例子：

```
void f(C& c);
vector<C> vc;
for_each(vc.begin(), vc.end(), f);
```

就上例来说代码是可以编译通过的，但是，当 f 是类 C 的成员函数时呢?

```
class C {
    public:
        void f();
};
```

此时我们就需要使用到类成员函数适配器了，在这时，mem_fun、mem_fun_ref 的区别在于 mem_fun 需要指针，而 mem_fun_ref 需要对象的引用。如上述例子中，应该使用 mem_fun_ref。

```
for_each(vc.begin(), vc.end(), mem_fun_ref(&C::f));
```

mem_fun 的用法类似，在此不赘述。由此可见，类成员函数适配器可以将一个不含参数的成员函数转换为一个一元函数，其中参数类型为类本身。此外，它也可以将一个一元成员函数转换为一个二元函数。

mem_fun 和 mem_fun_ref 的返回类型不仅仅只有 mem_fun_t 和 mem_fun_ref_t，而是根据转换后的函数类型不同而有所不同，所有的返回类型如表 B-1 所示。

表 B-1　类成员函数适配器的返回类型

	mem_fun	mem_fun_ref
一元函数	mem_fun_t	mem_fun_ref_t
二元函数	mem_fun1_t	mem_fun1_ref_t
const 修饰的一元函数	const_mem_fun_t	const_mem_fun_ref_t
const 修饰的二元函数	const_mem_fun1_t	const_mem_fun1_ref_t

现在我们来看 C++11 的新特性。我们前面已经介绍过了新的 bind 函数模板，它取代了原有的 bind1st 和 bind2nd。bind 不再需要 ptr_fun，因此 ptr_fun 被弃用。而对于 mem_fun 和 mem_fun_ref，C++11 提出了一个新的函数模板 mem_fn，它实现了 mem_fun 和 mem_fun_ref 的所有功能，而且更为强大。它不像 mem_fun 和 mem_fun_ref 那样只能处理一元函数或二元函数，它能够针对任意多个参数的函数进行转换；使用时，也不用再区分是指针还是一般对象。因此 mem_fun 和 mem_fun_ref 也被弃用，不仅如此，它们所有的返回类型也被弃用。另外，被弃用的还包括 unary_function 和 binary_function。

条目 8　动态异常声明（exception specification）

旧特性：异常声明 throw()

改动：有参数的异常声明被弃用，空异常声明 throw() 被 noexcept 取代。

函数可以通过异常声明来列出它直接或者间接可能抛出的异常。

形如 throw(T) 的异常声明成为动态异常声明。当一个函数抛出 E 类型的异常时，如果它的动态异常声明包含一个类型 T，且它的处理函数（handler）和类型 E 是匹配的，那么这个函数就允许 E 类型的异常。当抛出一个异常时，编译器会搜索处理函数，如果直到最外层的带有异常声明的代码模块都不允许此异常的话，那么如果是动态异常声明，就会调用 std::unexpected()。

实践证明，动态异常声明是没有价值的，只能给程序带来更多的开销。它主要的问题如下：

- C++ 的异常声明是一种运行时检查，而不是编译时，也就是说，在编译时不能确保所有的异常都能被处理，而运行时的失败模式（failure mode），也就是调用 std::unexpected() 并不能自身恢复。
- 运行时的检查会要求编译器生成更多代码，而这些代码会阻碍优化，增大运行时开销。
- 在泛型的代码中，很难预知在模板参数的操作过程中会抛出什么类型的异常，所以不太可能写出准确的异常声明。

因此，在 C++11 中，动态异常声明被弃用。

在动态异常声明中，作为唯一的例外而被认为有价值的是空异常声明，也就是 throw()。在实践中，只有两种异常的抛出确实是有用的：程序会抛出异常或者程序不会抛出异常。前者可以由完全省略异常声明来表示；后者则可以由 throw() 来表示。但是由于性能方面的考虑，还是很少被用到。

在 C++11 中，提出了一种新的异常声明 noexcept，关键字 noexcept 表示函数不会抛出异常，或者说异常不会被接获并处理。noexcept 异常声明除了有 noexcept 关键字的形式，还可以是 noexcept (constant-expression) 的形式，这里 constant-expression 要求可以被转换为 bool 类型。这样，noexcept 可以通过条件判断来决定函数是不是能够抛出异常。另外，noexcept 关键字的意义其实就等于 noexcept(true)。当用 noexcept 修饰的函数，也就是不允许抛出异常的函数中抛出异常时，编译器会调用 std::terminate()。

与 throw() 不同，noexcept 不需要编译器生成额外的代码来进行运行时检查，而且使用上更为灵活，因此完全可以取代 throw()。

综上所述，由于含有参数的动态异常声明在实际使用中没有价值，而空动态异常声明 throw() 已被 noexcept 取代，所以动态异常声明，也就是 throw (type-id-listopt) 被弃用。

附录 C

编译器支持

C++11 是否能够在 20 世纪的第二个 10 年光芒依旧，必不可少地需要整个行业的生态环境的支持。这意味着一方面是 C++11 在学习使用上的闪光点深入人心，而另外一方面，则是有广泛的编译器支持。

如同我们在第一章中提到的，事实上，大多数编译器组织或厂商都在着手支持 C++11。但 C++11 的特性非常多，以至于编译器组织或厂商通常需要若干个版本才能完全支持。在本书编写时，地球上的所有编译器都还未能完全地支持所有的 C++11 特性。不过这样的状况很快就会得到改变。一些开源的编译器项目，比如 GCC 以及 Clang，应该在不久的时间内即将成为第一个完全支持 C++11 的编译器（现在从我们得知的情况看来，GCC 可能会最早完成）。而商业编译器则相对会慢一些，而且是否完整支持 C++11 有时候也会依据客户需要而定。

在 IBM Power 平台上，最为常用的编译器是 IBM 的 XL C/C++ 及 GCC。截止本书完成，IBM 的 XL C/C++ 编译器的最新版本是 XL C/C++ V12.1，可以用于 AIX 以及 Linux 平台。XL C/C++ V12.1 支持了最为核心的特性，包括了：auto、c99、常量表达式 constexpr、decltype、委托构造函数、显式类型转换 operator、扩展的 friend 声明、外部模板、内联名字空间、long long、追踪返回类型的函数声明、右值引用以及移动语义、右尖括号、静态断言、强类型枚举、变长模板等。

在 XL C++12.1 中（请参见 http://www-01.ibm.com/software/awdtools/xlcpp/），程序员可以通过选项 -qlanglvl=extended0x 来开启对 C++11 大部分特性的支持，-qwarn0x 选项用来诊断 C++11 与 C++98 有区别的代码。此外，程序员还可以在 XLC++ 中仅仅通过一些子选项开启某一个 C++11 的功能，详细如表 C-1 所示。

表 C-1　IBM XL C/C++ 中有关于 C++11 的选项

IBM XL C/C++ 编译器选项	说　明
-qlanglvl=[no]autotypededuction	支持 C++11 中 auto 特性
-qlanglvl=[no]constexpr	支持 C++11 的常量表达式类型
-qlanglvl=[no]decltype	支持 decltype
-qlanglvl=[no]delegatingctors	支持委托构造函数
-qlanglvl=[no]c99longlong	支持 long long 数据类型
-qlanglvl=[no]inlinenamespace	支持内联名字空间

（续）

IBM XL C/C++ 编译器选项	说　明
-qlanglvl=[no]rvaluereferences	支持右值引用
-qlanglvl=[no]static_assert	支持 static assert
-qlanglvl=[no]variadic[templates]	支持变参模板

此外，一如既往，IBM 为所发布的特性都提供了良好的文档支持（请参见 http://pic.dhe.ibm.com/infocenter/lnxpcomp/v121v141/index.jsp）。

而在 x86 及 x86_64 平台上，编译器在 C++11 的支持上则呈现了百花齐放的状态。从商业编译器上讲，主要是 Intel 的 Intel C/C++ 编译器及微软的 MSVC 对 C++11 做了大量的支持。两者最新版本分别为 Intel C/C++ V13 以及 MSVC 2012（本书截稿时还没有正式发布）。

而在开源编译器上，GCC 及基于 llvm 的 clang 则同样站在 C++11 支持的最前列。GCC 对 C++11 的支持在 http://gcc.gnu.org/projects/cxx0x.html 可以找到，同样，clang 对 C++11 的支持可以从 http://clang.llvm.org/cxx_status.html 上找到。可以看见，两款编译器除了依赖于底层的并行特性和少量未完成的特性外，大部分 C++11 的特性都已经得到了支持。

虽然两款编译器都实现了极高的 C++11 支持度，不过两者现在也并未默认开启 C++11 编译支持。程序员可以使用 -std=c++11 来打开 C++11 模式，而选项 -std=gnu++11 可以同时支持 C++11 和 GNU 的扩展功能。

注意　可能读者对 clang 编译器还不是非常了解。不过在我们的使用中，clang++ 编译器则表现了很好的实用性，clang++ 基本上兼容了所有的 g++ 的编译选项，其错误输出在 shell 的支持下能够显示颜色，所以显得更加友好。有一些 Linux 的发布版中，我们已经看到使用 clang 代替 gcc 作为默认编译器的状况。

一些 GCC 中其他 C++11 相关选项则如表 C-2 所示。

表 C-2　GCC 一些与 C++11 有关的编译选项

GCC 编译器选项	说　明
-fabi-version=6	增强 abi 对 C++11 中限定范围的 enum 类型提升的支持
-fconstexpr-depth=n	设置 C++11 常量表达式的计算层数
-Wnarrowing	按 C++11 要求，对数据截断提供诊断信息
-Wc++11-compat	对 C++11 与 C++98 有区别的地方报错
-Wzero-as-null-pointer-constant	当数字 0 作为空指针使用时报错，C++11 中空指针是 nullptr

如第一章所描述的，在本书编写时，我们主要使用了 xl c/c++、gcc 和 clang 三种编译器。因此对其状态也较为熟悉。其他的，比如跟 Intel 编译器同样使用 EDG（Edison Design Group，一个专业的编译器前端厂商）前端的 HP C/aC++ 编译器、Comeau 编译器，以及

Borland/CodeGear 的 C++Builder 也都或多或少地加入了部分 C++11 的支持。

事实上，读者可以通过网页 http://wiki.apache.org/stdcxx/C++0xCompilerSupport 来获知主流编译器组织或厂商对 C++11 编译器的支持情况。这是一张横向比较的表格，如果读者想使用的 C++11 特性不在你的编译器包含之中的话，那么你应该写信催促一下开发者了。

附录 D

相关资源

在本书的编写过程中，作者参考了大量的资料。这些资料主要是一些特性的草案，以及一些源自网络的资源。前者往往通过草案的提出、讨论、修改、决议等各方面揭示了 C++ 特性发展演化的过程的所有情况，而后者则在对标准的阐释、辩析、理解上对本书的编写起了很大的帮助。同样，我们将一些资源罗列出来，以供试图了解 C++ 发展或者仅仅是由于本书未能解除心中疑惑的读者使用。

D.1 C++11 特性建议稿

所有关于 C++ 特性的建议案都在 WG21 的文档库中管理，其链接为：http://www.open-std.org/jtc1/sc22/wg21/docs/papers/，在 WG21 的文档库中，所有的文档都是按时间排列的。这些文档最终演化为 C++ 标准的一部分。表 D-1 则按照主题的方式将这些文档串联起来。由于文档数量较大，而且 WG21 也不时会改变文档的存储路径，所以我们建议读者通过搜索的方式来寻找需要的文档，即在 Google 中输入关键字“WG21”以及文档编号，如“N1377”。一般我们就可以获得该文档的有效 URL 路径。

表 D-1 按主题排列的 C++11 特性建议稿

主　题	英文主题（topic）	文档编号
右值引用	A Proposal to Add an Rvalue Reference to the C++ Language	N1377，N1385，N1690，N1610，N1770，N1855，N1952，N2118
静态断言	static_assert	N1381，N1604，N1617，N1720
模板别名	Template aliases for C++	N1406，N1449，N1451，N1489，N2112，N2258
外部模板	Extern template	N1448，N1960，N1987
*this 的移动语义	Extending Move Semantics To *this (Revision 2)	N1784，N1821，N2377，N2439
通过右值引用初始化类对象	Clarification of Initialization of Class Objects by rvalues	N1610
变长模板	Variadic Templates	N1483，N1603，N1704，N2080，N2152，N2191，N2242
扩展变长模板的参数	Extending Variadic Template Template Parameters	N2488，N2555

（续）

主　题	英文主题（topic）	文档编号
强类型枚举	Strongly Typed Enums	N1513，N1579，N1719，N2213，N2347
枚举类的前置声明	Forward Declaration of Enums	N2499，N2568，N2678，N2764
扩展的 friend 声明	Extended friend Declarations	N1520，N1616，N1722，N1791
泛化的常量表达式	Generalized Constant Expressions	N1521，N1972，N1980，N2116，N2235
一些关于 C99 宏定义	Synchronizing the C++ preprocessor with C99 : variadic macros, empty macro argument, concatenation of mixed char and wchar literals	N1545，N1566，N1653
对齐支持	Adding Alignment Support to the C++ Programming Language	N1546，N1877，N1971，N2140，N2165，N2252，N2301，N2341
条件支持的行为	"Conditionally-Supported Behavior"	N1564，N1627
将未定义行为变为可诊断的错误	Changing Undefined Behavior into Diagnosable Errors	N1727
增加 long long 类型	Adding the long long type to C++	N1565，N1693，N1735，N1811
扩展的整型	Adding extended integer types to C++	N1746，N1988
委托构造函数	Delegating Constructors	N1581，N1618，N1895，N1986
新字符类型 char16_t 和 char32_t	New Character Types in C++ char16_t,char32_t	N1628，N1823，N1955，N2018，N2149，N2249
右尖括号	Right Angle Brackets	N1649，N1699，N1757
由初始化表达式进行类型推导	Deducing the type of variable from its initializer expression	N1721，N1794，N1894，N1984
	A Proposal to Restore Multi-declarator auto Declarations	N1737
auto 的语法	The Syntax of auto Declarations	N2337，N2546
追踪返回类型	New function declaration syntax for deduced return types	N2445，N2541
	A finer-grained alternative to sequence points	N1944，N2052，N2171，N2239
__func__ 预定义标识符	Proposed addition of __func__ predefined identifier from C99	N1970，N2202，N2251，N2340
POD	PODs unstrung	N2062，N2102，N2172，N2230，N2294，N2342
在 thread join 的时候复制异常	Propagating exceptions when joining threads	N2096，N2179
decltype	Decltype	N2115，N2343
Decltype 及调用表达式	Decltype and Call Expressions	N3276
扩展的 sizeof	Extending sizeof	N2150，N2253
显示缺省和删除的函数	Defaulted and Deleted Functions	N2210，N2326，N2346
	Not so trivial issues with trivial	N2762

（续）

主　题	英文主题（topic）	文档编号
lambda 函数	(monomorphic) lambda expressions and closures for C++	N1968，N2329，N2413，N2487，N2529，N2550
lambda 函数的正确性	Constness of lambda functions	N2561，N2658
指针空值 nullptr	A name for the null pointer: nullptr	N1488，N1601，N2214，N2431
	Core issue 654:convertibility of 0-literal	N2656
继承构造函数	Inheriting Constructors	N1898，N2119，N2203，N2254，N2376，N2438，N2512，N2540
显式转换操作符	Explicit Conversion Operators	N1592，N2223，N2333，N2380，N2437
原生 Unicode 字符串字面量	Raw.unicode String Literals	N2053，N2146，N2295，N2384，N2442
字符串中的 unicode 字符	Universal Character Names in Literals	N2170
名字空间的联合	Namespace Association ("Strong Using")	N1526，N2013，N2331，N2535
非受限联合体	Unrestricted Unions	N2248，N2430，N2544
原子操作	Atomic operations with multi-threaded environments	N2047，N2145，N2324，N2381，N2393，N2427
顺序及内存模型	Sequencing and the concurrency memory model	N2052，N2171，N2300，N2334，N2429
快速退出程序	Abandoning a Process(at_quick_exit)	N2383，N2440
允许信号捕捉函数使用 atomic	Allow atomics use in signal handlers	N2459，N2547
多线程库	A Multi-threading Library for Standard C++	N2320，N2447
UTF8 字面量	Unicode Strings UTF8 Literals	N2159，N2209，N2295，N2384，N2442
type_trait 及局部类	type_trait names Making Local Classes more Useful	N1427，N1945，N2187，N2402，N2635，N2657
初始化列表	Initializer lists	N1509，N1890，N1919，N2100，N2215，N2385，N2531，N2672
线程局部存储	Thread-Local Storage	N1874，N1966，N2147，N2280，N2545，N2659
数据相关的排序：原子操作与内存模型	C++ Data-Dependency Ordering: Atomics and Memory Model	N2492，N2556，N2664
数据相关的排序：函数	C++ data-dependency ordering: function annotation	N2361，N2493
动态初始化及并行	Dynamic initialization and concurrency	N2148，N2325，N2382，N2444，N2513，N2660
最小垃圾回收支持	Minimal Support for Garbage Collection and Reachability-Based Leak Detection	N2481，N2527，N2585，N2586，N2670
SFINAE	Solving the SFINAE problem for expressions	N2634
双向栅栏	Proposed text for bidirectional fences	N2633，N2731，N2752
成员快速初始化	Member Initializers	N1959，N2354，N2426，N2756

（续）

主　题	英文主题（topic）	文档编号
一些概念	Concepts (unified proposal)	N2042，N2081，N2193，N2307，N2398，N2421，N2501，N2617，N2676，N2710，N2741
一些概念	Named requirements for C++0x concepts	N2581，N2780
基于范围的 for	Wording for range-based for-loop (revision 3)	N1868，N1961，N2049，N2196，N2243，N2394
通用属性	General Attributes for C++	N2224，N2236，N2379，N2418，N2466，N2553，N2751，N2761
用户自定义字面量	Extensible Literals	N1511，N1892，N2282，N2378，N2747，N2750，N2765
显式重载	Explicit Virtual Overrides	N2928
允许移动构造函数抛出异常	Allowing move constructors to throw [noexcept]	N3050
默认移动构造函数	Defining move special member functions	N3053
强 CAS 操作	Strong compare and exchange	N27485

值得注意的是，编号较小的文档通常会对相关主题的描述较多，而编号较大的文档则通常着重改善之前的特性，以及着重于如何对 C++ 标准进行修改。因此最后一篇文档常常未必是读者需要的。

D.2　其他有用的资源

在本书写作时，关于 C++11 的资源还算不上丰富（相比于它的前任 C++98/03 而言）。因此，大多数的其他资源都来自于网络。网络的缺点是链接常常会失效。不过在本书新鲜出炉的时候，相信这些链接还是有效的。

特性介绍类

- http://en.wikipedia.org/wiki/C%2B%2B11，这是 wikipedia 中关于 C++11 介绍。比较全面，如果想对所有特性进行快速学习，wiki 总是不容错过的。
- http://www.stroustrup.com/C++11FAQ.html，C++ 之父 Bjarne Stroustrup 关于 C++11 的介绍。Bjarne 会不时更新一些部分。该页面上也有中文翻译的链接。不过看起来还有不少特性 Bjarne 还没来得及写。
- *C++ Primer Plus Sixth Edition*，这是 Primer Plus 系列的第六版，其中对 C++11 有一些介绍。本书针对的是 C++ 初学者，而中文版现在也已经问世了。
- http://www.cprogramming.com/c++11/what-is-c++0x.html，这是 Alex Allain 关于 C++11 的一篇综述。不过文章末尾有一些链接，则是 Alex Allain 对 C++11 一系列的特性分别详述的文章。而且最为难能可贵的是，每一篇都保持了高质量。

❑http://zh.wikipedia.org/wiki/C++0x，这是 wiki 中文中对 C++11 的描述。跟英文版一样，保持了很好的特性分类。这也是我们推荐的为数不多的中文资源。

技术参考类

❑http://en.cppreference.com/w/，cppreference 应该是所有同类型网站中我们最喜欢的。通过搜索，读者可以找到任何关于 C++ 的特性描述、库工具等，而且大多数带有简单易懂的例子。

❑http://stackoverflow.com/，可以肯定的是，在 stackoverflow 的网上，常会有世界级的 C++ 专家出没。任何困难的 C++11 问题，基本上都可以在 stackoverflow 上搜索到相关的答案。

其他

❑http://www.open-std.org/jtc1/sc22/wg21/，WG21 的主页，读者可以在这里找到 C++ 标准委员会的邮件、文档、会议等各种信息，甚至是一些标准的草稿。

❑Adve, Gharachorloo, Shared Memory Consistency Models: A Tutorial，这是一篇介绍内存模型的非常好的论文，作者通过对多个硬件平台的比较，总结归纳了软硬件平台的内存一致性实现的方式。读者应该很容易搜索到。